

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



### THESIS

**APPLICATION OF FAULT-TOLERANT COMPUTING  
FOR SPACECRAFT USING COMMERCIAL-OFF-THE-  
SHELF MICROPROCESSORS**

by

Susan E. Groening  
and  
Kimberly Davenport Whitehouse

June 2000

Thesis Co-Advisors:

James B. Michael  
Alan A. Ross

Approved for public release; distribution is unlimited

DTIC QUALITY INSPECTED 4

20000807 074

<b>REPORT DOCUMENTATION PAGE</b>		Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> June 2000	<b>3. REPORT TYPE AND DATES COVERED</b> Master's Thesis
<b>TITLE AND SUBTITLE :</b> Application Of Fault-Tolerant Computing For Spacecraft Using Commercial-Off-The-Shelf Microprocessors		<b>5. FUNDING NUMBERS</b>	
<b>5. AUTHOR(S)</b> Groening, Susan E. and Whitehouse, Kimberly Davenport			
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Postgraduate School Monterey, CA 93943-5000		<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> N/A		<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>	
<b>11. SUPPLEMENTARY NOTES</b> The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution is unlimited		<b>12b. DISTRIBUTION CODE</b>	
<b>13. ABSTRACT (maximum 200 words)</b> Low availability, high cost, and poor performance of radiation hardened (rad-hard) equipment has driven the market to rely on commercial-off-the-shelf (COTS) equipment for the computing needs of today's spacecraft. This thesis describes the tailoring of a COTS embedded real-time operating system and design of a human-computer interface (HCI) for a triple modular redundant (TMR) fault-tolerant microprocessor for use in space-based applications. One disadvantage of using COTS hardware components is their susceptibility to the radiation effects present in the space environment, and specifically, radiation-induced single-event upsets (SEUs). In the event of an SEU, a fault-tolerant system can mitigate the effects of the upset and continue to process from the last known correct system state. The TMR basic hardware design used for this research is an acceptable fault-tolerant design candidate for the main processor for space-based applications. We found that a COTS embedded real-time operating system could be tailored to support the TMR hardware. The HCI accepts serial data from the TMR, correctly identifies the source of the error, allows for processor mode selection and provides system- and board-level reset capabilities. The tailored operating system combined with the HCI is a viable software implementation to support hardware-based fault-tolerant computing in a space environment.			
<b>14. SUBJECT TERMS</b> Fault Tolerance, Embedded Operating System, Human-Computer Interface, Triple Modular redundant hardware, Spacecraft Design			<b>15. NUMBER OF PAGES</b> 167
			<b>16. PRICE CODE</b>
<b>17. SECURITY CLASSIFICATION OF REPORT</b> Unclassified	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> Unclassified	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> Unclassified	<b>20. LIMITATION OF ABSTRACT</b> UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**APPLICATION OF FAULT TOLERANT COMPUTING FOR SPACECRAFT  
USING COMMERCIAL-OFF-THE-SHELF MICROPROCESSORS**

Susan E. Groening  
Lieutenant, United States Navy  
B.A., University of Florida, 1989  
and  
Kimberly Davenport Whitehouse  
Captain, United States Marine Corps  
B.S., University of Florida, 1990

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

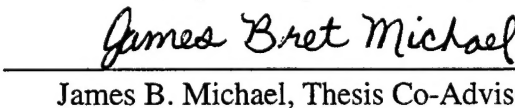
**NAVAL POSTGRADUATE SCHOOL  
June 2000**


Authors:

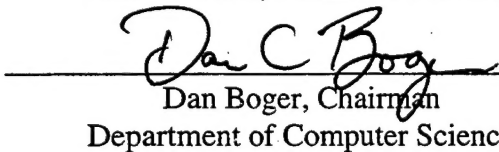
  
Susan E. Groening

  
Kimberly Davenport Whitehouse

Approved by:

  
James B. Michael, Thesis Co-Advisor

  
Alan A. Ross, Thesis Co-Advisor

  
Dan Boger, Chairman  
Department of Computer Science



THIS PAGE INTENTIONALLY LEFT BLANK

## ABSTRACT

Low availability, high cost, and poor performance of radiation hardened (rad-hard) equipment has driven the market to rely on commercial-off-the-shelf (COTS) equipment for the computing needs of today's spacecraft. This thesis describes the tailoring of a COTS embedded real-time operating system and design of a human-computer interface (HCI) for a triple modular redundant (TMR) fault-tolerant microprocessor for use in space-based applications. One disadvantage of using COTS hardware components is their susceptibility to the radiation effects present in the space environment, and specifically, radiation-induced single-event upsets (SEUs). In the event of an SEU, a fault-tolerant system can mitigate the effects of the upset and continue to process from the last known correct system state. The TMR basic hardware design used for this research is an acceptable fault-tolerant design candidate for the main processor for space-based applications. We found that a COTS embedded real-time operating system could be tailored to support the TMR hardware. The HCI accepts serial data from the TMR, correctly identifies the source of the error, allows for processor mode selection and provides system- and board-level reset capabilities. The tailored operating system combined with the HCI is a viable software implementation to support hardware-based fault-tolerant computing in a space environment.

THIS PAGE INTENTIONALLY LEFT BLANK

## TABLE OF CONTENTS

<b>I.</b>	<b>INTRODUCTION.....</b>	<b>1</b>
<b>A.</b>	<b>THE SPACE ENVIRONMENT .....</b>	<b>1</b>
1.	Gravity.....	1
2.	Atmosphere .....	1
3.	Vacuum.....	2
4.	Micrometeoroids and Space Junk.....	3
5.	Radiation .....	3
6.	Charged Particles .....	3
<b>B.</b>	<b>BACKGROUND.....</b>	<b>4</b>
1.	Radiation Hardened Devices .....	4
2.	Fault Tolerance.....	5
<b>C.</b>	<b>PURPOSE .....</b>	<b>9</b>
<b>D.</b>	<b>THESIS ORGANIZATION .....</b>	<b>10</b>
<b>II.</b>	<b>OPERATING SYSTEM SELECTION.....</b>	<b>13</b>
<b>A.</b>	<b>OPERATING SYSTEM SELECTION.....</b>	<b>13</b>
1.	Definitions .....	13
2.	Buy vs. Build .....	14
3.	Characteristics of a Real-time Operating System .....	15
4.	Criteria for Selection of a Real-time Operating System .....	17
a.	<i>Processor Support</i> .....	18
b.	<i>Portability</i> .....	18
c.	<i>Scalability</i> .....	20
d.	<i>Multiprocessor Support</i> .....	20
e.	<i>Extended Services</i> .....	21
f.	<i>Vertical Applications</i> .....	21
g.	<i>POSIX Compliance</i> .....	21
h.	<i>Language Support</i> .....	22
i.	<i>Development Environment</i> .....	23
j.	<i>Licensing and Cost</i> .....	23
<b>B.</b>	<b>OBSERVATIONS .....</b>	<b>24</b>
<b>III.</b>	<b>HUMAN COMPUTER INTERFACE DESIGN .....</b>	<b>27</b>
<b>A.</b>	<b>OVERVIEW .....</b>	<b>27</b>
<b>B.</b>	<b>NEEDS ANALYSIS .....</b>	<b>30</b>
<b>C.</b>	<b>USER ANALYSIS .....</b>	<b>31</b>
<b>D.</b>	<b>TASK ANALYSIS.....</b>	<b>32</b>
<b>E.</b>	<b>CONCEPTUAL DESIGN AND VISUAL MODEL.....</b>	<b>34</b>
1.	Reviewer Analysis .....	36
2.	Resultant Changes.....	39
3.	Observation.....	40

F.	PROTOTYPE .....	40
IV.	HUMAN COMPUTER INTERFACE RAPID PROTOTYPE DEVELOPMENT AND TESTING .....	43
A.	PROTOTYPE DEVELOPMENT .....	43
1.	Rapid Prototype Revision .....	43
a.	UART .....	44
b.	FPGA .....	44
c.	First-In-First-Out (FIFO) registers .....	45
d.	TMR processors .....	45
e.	Voter .....	45
f.	EPROM .....	45
2.	Rapid Prototype Design .....	49
3.	Modal Dialog Boxes .....	50
4.	Error Detection .....	56
5.	HCI Testing .....	59
B.	DISCUSSION .....	63
1.	Test Results .....	63
2.	Value of Storing Register Contents .....	63
V.	BOARD SUPPORT PACKAGE .....	65
A.	BACKGROUND .....	65
B.	CREATING A BSP .....	67
1.	Basis of Development .....	69
2.	BSP Pre-kernel Initialization Code .....	71
3.	Start a Minimal VxWorks Kernel and Add the Basic Drivers .....	75
4.	Start the Target Agent and Connect the Tornado Development Tools .....	76
5.	Complete the BSP .....	77
6.	Generate a Default Project for the New Project Facility .....	77
C.	DISCUSSION .....	78
D.	LESSONS LEARNED .....	78
VI.	CONCLUSION AND FUTURE DIRECTIONS .....	81
A.	CONCLUSION .....	81
B.	FUTURE DIRECTIONS .....	82
	APPENDIX A. TMRINTERFACECLASS.JAVA .....	85
	APPENDIX B. READERTHREAD.JAVA .....	103
	APPENDIX C. BUILDDATAATHREAD .....	109
	APPENDIX D. FIFOBUFFER.JAVA .....	125
	APPENDIX E. TMRTESTPANEL.JAVA .....	129
	APPENDIX F. BUILDING A PROJECT FROM A NEW BSP .....	141
	LIST OF REFERENCES .....	147

INITIAL DISTRIBUTION LIST .....	149
---------------------------------	-----

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF FIGURES

Figure 1-1.	TMR Concept.....	8
Figure 3-1.	Flow Chart for User Scenarios 1 and 2 .....	38
Figure 4-1.	TMR Hardware Design .....	44
Figure 4-2.	Revised Flow Chart .....	48
Figure 4-3.	“Welcome” Dialog Box.....	51
Figure 4-4.	“Select Processor Mode” Dialog Box .....	51
Figure 4-5.	“Verify Processor Mode” Dialog Box.....	52
Figure 4-6.	“Execute Program” Dialog Box .....	52
Figure 4-7.	“TMR Testbed Main Screen” .....	53
Figure 4-8.	“Reset Confirmation” Dialog Box .....	54
Figure 4-9.	“Reset” Dialog Box .....	54
Figure 4-10.	“Save History Information” Dialog Box .....	55
Figure 4-11.	Array Comparison Logic to Determine Error Processor. ....	58
Figure 4-12a.	High Level Example of Test Data Stream After Error Detected.....	59
Figure 4-12b.	Individual Processor’s Data Stream .....	59
Figure 4-13.	Header Format .....	60
Figure 4-14.	FIFO Register Format for Each Processor .....	61
Figure 5-1.	Host Tools Communication .....	66
Figure 5-2.	Hardware Dependent and Independent Software .....	68
Figure 5-3.	Pre-kernel Initialization Sequence.....	71



THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF TABLES

Table 3-1.	Conceptual Design .....	35
------------	-------------------------	----

THIS PAGE INTENTIONALLY LEFT BLANK

## ACKNOWLEDGMENTS

Susan E. Groening:

Thank you, my Heavenly Father for your unending love, patience and guidance. Thank you, my loving husband, Rick, for your devotion and generosity when you had to carry it all. Thank you, my wonderful children, Matt, Zach, and Katie for your understanding when mom had to be at school. Thank you, my mother, Marjorie Owens, for making me believe in myself. Thank you, Kim, for being not only a thesis partner, but a friend.

A sincere thank you to Dr. Bret Michael, Dr. Alan Ross, and LCDR Chris Eagle for their continuous guidance and support.

Kimberly D. Whitehouse:

First and foremost, I would like to give thanks and praise to my Heavenly Father. "Trust in the LORD with all your heart and lean not on your own understanding; in all your ways acknowledge him, and he will make your paths straight." (Proverbs 3:5-6) I would like to thank Professor Michael and Professor Ross for their guidance and patience during the course of this thesis research. A special thank you to LCDR Chris Eagle (a.k.a. the human compiler) who was always helpful, even after answering the millionth programming question. To my thesis partner, Susan: thank you for keeping me motivated, always being there to encourage me, and most importantly, for being a wonderful friend! I would like to thank my parents, Bobby and Janice Davenport for all their support and love. I would like to thank my children, Zachary, Alexis, Alison, Caroline and Jacob for their love, patience, and understanding during all those long days and nights while I was studying. Finally, I would like to thank my husband and my best friend, Thomas, for his selflessness, love, and understanding.

THIS PAGE INTENTIONALLY LEFT BLANK

# **I. INTRODUCTION**

## **A. THE SPACE ENVIRONMENT**

The Earth, the Sun, and the cosmos combined offer six different types of challenges that affect the design of spacecraft. The challenges to design include gravity, atmosphere, micrometeoroids and debris, vacuum, radiation, and charged particles (Sellers 65). Designers of computers that will operate in space must take into account this hostile environment and its affect on proper spacecraft operation. Though the focus of our research is protecting against the effects of radiation and charged particles, for completeness, each of the hazards is discussed briefly here.

### **1. Gravity**

The Earth's gravitational pull dominates the objects that are close to it, but as spacecraft get further and further away, the gravitational affects of the moon and the sun begin to have an influence on the orbits of spacecraft. The size and shape of a spacecraft's orbit is dictated by gravity. Booster rockets must first overcome Earth's gravity to propel the craft into space. Once it is in orbit, gravity determines the amount of propellant its engines must use to move between orbits or link up with other spacecraft (Sellers 66).

### **2. Atmosphere**

The Earth's atmosphere affects a spacecraft in low-Earth orbit (below 600 kilometers) in two ways:

-Drag, caused by atmospheric density, shortens orbit lifetimes. For example, drag is used to slow the U. S. Space Shuttle from an orbital velocity of over twenty-five times the speed of sound, to a runway landing at 225 m.p.h. Drag can cause a low-orbiting spacecraft to be pulled back into the Earth's atmosphere.

-Atomic oxygen (O) is caused by oxygen ( $O_2$ ) being ionized by the Sun's radiation.  $O_2$ , when ionized, splits into two (O) ions, atomic oxygen. These ions combine with the metals of the spacecraft and results in oxidation, commonly known as rust. Oxidation degrades spacecraft surfaces and reduce their lifetime (Sellers 67).

### **3. Vacuum**

Beyond the Earth's atmosphere is the cold vacuum of space. This vacuum creates three potential problems for spacecraft:

-Outgassing, a release of gasses from spacecraft materials, which is usually not a problem. However, the released gasses can coat delicate sensors, such as lenses. When that happens, outgassing can be considered destructive.

-Cold welding, which occurs between metal parts that have very little separation between them. On Earth, air in any tiny gap allows the parts to move. However, in space, the vacuum removes this air strip and the two metal parts fuse together. Designers must select appropriate lubricants so they will not evaporate or outgas.

-Managing heat transfer by radiation. As radiation does not need a solid or fluid medium, it is the primary method of moving heat into and out of the spacecraft. The Sun's radiation can be destructive to continuously exposed components if they are not properly cooled.

(Sellers 68).

#### **4. Micrometeoroids and Space Junk**

Space is full of natural debris such as dust, meteoroids, chunks of asteroids, and manmade items like old booster segments and retired satellites. The likelihood of being struck by an object that is only 1 millimeter in diameter is very slight (less than 1 in 1000) over the planned mission of the spacecraft. But if a satellite is struck by a small particle travelling at 7000 meters/second, it will create more energy than a rifle bullet and can be destructive (Sellers 70).

#### **5. Radiation**

The radiation environment above the Earth's protective atmosphere is harsh. The Sun's X-rays and gamma rays bombard spacecraft causing problems with overheating, degradation, and damage to surfaces and electric components. In order to protect the spacecraft against the harmful effects of radiation, the electronic components must be shielded or hardened. (Sellers 71).

#### **6. Charged Particles**

Charged particles present perhaps the most dangerous aspect of the space environment. Three primary sources for these particles include the solar wind and flares, Galactic cosmic rays, and the Van Allen radiation belts. Regardless of their origin, charged particles can harm spacecraft in three ways:

- Spacecraft charging occurs when charges build up on different parts of the spacecraft as it moves through areas of concentrated charged particles. Once the charge



builds up, discharge can occur with disastrous effects such as damage to surface coating, degrading of solar panels, or permanently damaging electronics.

-Sputtering is caused by the spacecraft being constantly bombarded by atomic particles. Over time, sputtering can damage thermal coatings and sensors.

-Single event effects. A single charged particle can penetrate deep into the interior of the spacecraft and disrupt its electronics. Each disturbance is known as a single event effect. (Sellers 72).

The methods, systems, and ongoing research that serve to protect against the hazards of gravity, atmosphere, vacuum are outside the scope of this thesis. Our focus is on masking or mitigating the effects of radiation and charged particles on electronics; therefore, a thorough discussion of these effects is included. Our research addresses a method to provide a spacecraft's processor with continued capability to operate when exposed to radiation and/or charged particles.

## **B. BACKGROUND**

### **1. Radiation Hardened Devices**

In the past, the most common approach to achieve survivability from the effects of radiation and charged particles was through the use of radiation hardened (rad-hard) devices. Rad-hard devices, used in conjunction with redundant hardware or error detecting/correcting codes, provide protection for the satellite's computer against the effects of radiation and charged particles.

Over the past 10 years, the number of suppliers of rad-hard devices has decreased and the prices of the devices have risen as many manufacturers are switching from the fabrication of rad-hard devices to the more lucrative commercial-off-the-shelf (COTS) non-rad-hard devices. This shift in availability poses significant design issues for Department of Defense (DoD) and commercial satellite ventures. As the rad-hard components are becoming increasingly cost prohibitive, the technology lag that results from the long procurement process to obtain the components prevents cutting-edge implementations and retards development cycles.

Due to the high cost and lengthy procurement cycle of rad-hard devices, current research focuses on the use of non-rad-hard hardware to provide survivability from the radiation effects of space. An alternative to rad-hard SEU protection is redundant hardware operating in lockstep, in conjunction with a specially designed operating system, to provide the benefits of rad-hard components at a greatly reduced cost.

## **2. Fault Tolerance**

The concept of computer fault tolerance, the ability of a system to continue to perform its intended purpose despite a hardware and/or software error, is not new and was used in the earliest computers. The EDVAC, designed in 1949, was equipped with redundant Arithmetic Logic Units (ALUs) in order to detect faulty processing of algorithms. Hardware components at that time were known to be unreliable and prone to failure (Storey 113). Today, fault tolerance techniques typically employ some combination of redundancy in the hardware or the software.

Faults are usually characterized by their nature, duration, or extent. The nature of a fault is related to its cause, either random or systemic. The usual cause of a random fault is a hardware failure. Systemic faults occur as a result of a design flaw, either in the requirements specification, system design, or the implementation of the design in software or hardware. Faults can be described by their duration as well: permanent, transient, or intermittent. A fault that remains until some action is taken to correct it is referred to as a permanent fault. Transient faults sporadically occur and then disappear. A frequent cause of transient faults can be the effects of atomic particles hitting the memory chip. These types of faults can change the state of the computer without causing lasting damage to the system. An intermittent fault occurs, disappears, and then reoccurs, for example, a faulty solder joint. (Storey 114).

In a space environment, there are three types of radiation effects that can affect integrated circuits and cause faults: Total Dose Effects, Dose Rate Effects, and Single Event Effects. There are four sub-categories of Single Event Effects; Single Event Upset (SEU), Single Event Latchup (SEL), Single Event Gate Rupture (SEGR), and Single Event Burnout (SEB) (Payne 2). All of these radiation types, except SEU, are destructive to integrated circuitry.

An SEU can be described as a type of transient fault. Because an SEU is simply a bit flip caused by an ionized charge in a circuit, its effects will manifest as an erroneous instruction or data. As this type of fault is not physically destructive to the system, recovering from an SEU, which means catching the bit flip and correcting it, can be

achieved without rad-hard devices. This type of fault tolerance can be implemented with configurable COTS software and hardware.

This research addresses the ability to obtain fault tolerance for a space system using COTS software and redundant hardware created from non-rad-hard COTS devices. Our thesis extends the research performed by John C. Payne, Jr., Naval Postgraduate School, December 1998. LT Payne developed the design of a testbed for assessing techniques used to resolve SEU-induced faults. The testbed computer employed a three-CPU triple modular redundant (TMR) design. LT Payne selected the R3081 processor, a COTS, single chip, RISC architecture machine with a 32-bit multiplexed address/data bus.

The basic concept of TMR is fairly simple. It requires the triplication of the hardware and performing a majority vote to determine the output of the system as shown in Figure 1 (Payne 33).

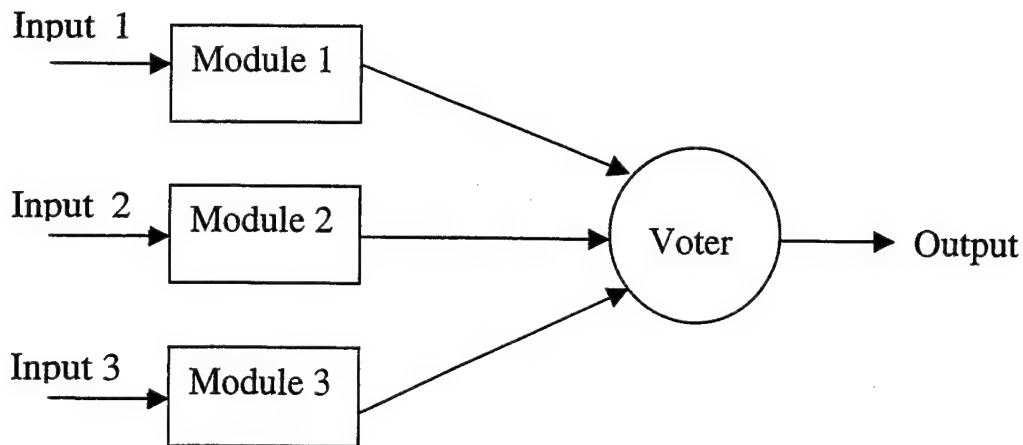


Figure 1-1. TMR Concept

This technique is considered to be a form of passive hardware redundancy in that it masks the occurrence of faults. Fault tolerance is achieved through the use of majority voting techniques without the need for fault detection or system recovery. If one of the modules becomes faulty, the two remaining modules, which are assumed to be fault free, mask the fault when the majority vote is performed (Storey 124). In a TMR system, an SEU could cause one processor to branch to a completely wrong address. That processor will continue to cause errors on all votes until it is reset to the same state of the two correct processors.

Mr. David Summers, Naval Postgraduate School, has shown that the design proposed by Mr. Payne is a good fault tolerant design candidate for the main processor for space-based applications. He has designed and fabricated a breadboard model of the system. Mr. Summer's research includes selection and programming of an FPGA (Field-Programmable Gate Array) for the voting and control logic. Additionally, he has

designed and overseen the fabrication of the main circuit board to hold the system components, as well as designed the memory space and interfaces.

### **C. PURPOSE**

In order to achieve fault tolerance for space-based microprocessors, our research will provide a design and implementation of software for the TMR microprocessor system. The software will be designed specifically for use in testing and evaluation of the TMR hardware in a laboratory environment, as well as in space-based applications to address the effects of SEU on non-rad-hard devices.

Our research will produce two distinct, but highly coupled components: first, a tailored COTS-based embedded operating system programmed to properly handle interrupts initiated by the TMR circuitry and second, a human computer interface (HCI) to the TMR system for testing and evaluation. The TMR voter will detect a bit flip in one of the three processors and signal an interrupt to the operating system. The operating system will process the interrupt by saving the register contents through the voter, thereby correcting the fault.

The HCI provides the interface to the TMR system so that users can control the operation and examine the results of errors and faults. The HCI has the functionality to load the application to be tested onto the TMR, start the application running, and throughout run-time, pause the TMR to permit examination of the states of the processors. Following an interrupt, the HCI translates the raw data received from the

TMR and extracts information about the processors throughout the application run-time. The HCI also logs the error data into a database and displays the errors to the user. The HCI supports system- and board-level resets of the TMR during testing.

The complete system (the TMR, operating system, and HCI) will be tested in a cyclotron laboratory environment during follow-on research. A cyclotron is a type of particle accelerator that produces certain types of particles in a particular energy range. During testing, the cyclotron will be focused on a single processor of the TMR so that the particles it produces will induce bit flips in the radiated processor only. This will create Single Event Upsets to test the masking effectiveness of the system. In order to properly mask the fault, the hardware must recognize the error, the operating system will reload the correct register information, and the HCI will display the error.

We envision that follow-on research will involve placing the experimental TMR system onboard a satellite to study the effects of actual SEU in the space environment on the system. The benefit of such research would be the development of small, relatively economical satellites by showing COTS software products can support COTS hardware to provide fault tolerance as a feasible alternative to radiation-hardened devices in space applications.

#### **D. THESIS ORGANIZATION**

Chapter II describes the operating system selection process and the characteristics of the selected operating system. Chapter III is a description of the design and

implementation of the HCI, while Chapter IV contains a discussion of testing and analysis of the final HCI tools. Chapter V describes the tailoring of the operating system. The conclusions and recommendations for future research are presented in Chapter VI.



THIS PAGE INTENTIONALLY LEFT BLANK

## **II. OPERATING SYSTEM SELECTION**

### **A. OPERATING SYSTEM SELECTION**

An operating system is the most important program that runs on a computer. Every general-purpose computer must have an operating system to manage system resources (e.g., files and devices). Operating systems perform many tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on a disk, and controlling peripheral devices.

For networked or distributed systems, the operating system has even more responsibilities. It is like a traffic cop -- it makes sure that different programs, users, and computing platforms running at the same time do not interfere with each other. The operating system is responsible for all aspects of management including such duties as security and quality of service.

#### **1. Definitions**

Operating systems can be classified as follows:

- **Multi-user:** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- **Multiprocessing:** Supports running a program on two or more CPUs.
- **Multitasking:** Allows more than one program to run concurrently.
- **Multithreading:** Allows different parts of a single program to run concurrently.

- Real-time: Allows applications to meet critical deadlines. General-purpose operating systems, such as Windows NT and UNIX, are not real-time. Unlike a general-purpose operating system, real-time operating systems utilize preemptive priority-based scheduling, are relatively faster, and are small and configurable (i.e., have a micro-kernel architecture).

Operating systems provide a software platform, on top of which other programs, called application programs, can run. Generally, an application program must be written to run on top of a particular operating system. However, recent innovations in higher level languages permit some degree of portability (i.e., platform independence).

An embedded system is a specialized computer system that is part of a larger system or machine. Typically, an embedded system is housed on a single microprocessor board with the programs stored in Read-Only Memory (ROM).

## **2. Buy vs. Build**

For this project, an important consideration was whether to buy a COTS operating system or build a custom operating system. The source code of many real-time operating systems exceeds 11,000 lines of code (Hawley 2). Although the size of the image is more important than lines of code, specifying, designing, implementing and testing an operating system would have been unattainable within the time constraints of our research. More importantly, the primary focus of this research is achieving fault tolerance using the TMR. By maintaining the COTS vision for the development of the TMR system and buying a commercial, real-time operating system that has been tested by the

development company and other customers, the system development process is accelerated.

In order to select an operating system for this project, we investigated the functionality and performance of several COTS real-time operating systems.

### **3. Characteristics of a Real-time Operating System**

Real-time operating systems can be characterized as having the following properties:

- Determinism
- Responsiveness
- Reliability
- Stability

An operating system is considered to be deterministic to the extent that it performs operations at fixed, predetermined times or within predetermined time intervals. When multiple processes are competing for resources and processor time, no system will be fully deterministic. In a real-time operating system, process requests for service are dictated by external events and timings. The extent to which an operating system can deterministically satisfy requests depends on the speed with which it can respond to interrupts and whether the system has sufficient capacity to handle all requests for computational resources within the required amount of time (Stallings 430).

A related but distinct characteristic is responsiveness. Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt (Stallings 430).

Reliability is typically far more important for real-time systems than non-real-time systems (Stallings 431). Simple rebooting of the system may solve a transient failure in non-real-time situations (e.g., word processing). A processor failure in a multiprocessor non-real-time system may result in a reduced level of service until the failed processor is repaired or replaced. But a real-time system must respond to events in real time (e.g., storage of streaming satellite telemetry data). Therefore, when a fault occurs in a real-time system, the system must be able to continue to meet required deadlines.

Although the focus of our research is to provide an operating system and human-computer interface for a fault-tolerant hardware implementation, in general, a real-time system must be designed to respond to various failure modes. Stability is a characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible by meeting the most critical deadlines even when less critical deadlines are not always met (Stallings 431). A real-time system will attempt to either correct the problem or minimize its effects while continuing to run.

To address the above characteristics, current commercial real-time operating systems typically include the following features:

- Hard Tasking (i.e., continues to operate when tasks fail)
- Fast context switching
- Small size (with its associated minimal functionality)
- Ability to respond to external interrupts quickly
- Multitasking with interprocess communication tools such as semaphores, signals, and events

- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling based on priority
- Minimization of intervals during which interrupts are disabled
- Primitives to delay tasks for a fixed amount of time and to pause/resume tasks
- Special alarms and timeouts (Jun 4).

#### **4. Criteria for Selection of a Real-time Operating System**

Commercial-off-the-shelf RTOS environments emerged more than a decade ago. Today, there are many to choose from. In the past, product comparisons centered on how an RTOS minimized latencies. Because each product's target system requires real-time deterministic response, minimizing latencies associated with external interrupts, kernel services, and task switching was of primary importance.

Today performance is still important, although the improved cycle time of modern microprocessors has diminished the significance of cycle time in many cases. Other factors that should be considered when evaluating an RTOS include the following:

- Support for the target processor
- Portability to new processors
- Scalability to match varied application requirements
- Multiprocessor support
- Extended services such as network support
- Vertical application support (a vertical application is software for a specific industry or group of computer users who share a set of well-defined needs)

- Standards(e.g., POSIX (Portable Operating System Interface for UNIX)) compliance
- Language support
- Development environment (including integration of the tools)
- Licensing arrangements and price (Artysyn 2; Jun 8)

*a. Processor Support*

When selecting an operating system, the most important consideration is whether the candidate operating system will provide long-term use for not only the current processor, but also future ones. Learning to use a specific RTOS and developing applications requires a significant investment in both time and money. Ensuring reuse of the same environments on future products can be of paramount importance to developers. It is important not only to consider whether an RTOS supports the processors used for the current project, but also other processors supported by each RTOS because future projects may require the use of a different processor.

*b. Portability*

In the long term, the portability of an RTOS from one processor to another is significant because faster processors with increased functionality will inevitably emerge. Although an RTOS may change considerably due to such factors as revolutionary hardware architectures and software development support, knowing the amount of time an RTOS vendor spent porting to the most recent processor

can give an indication of how quickly the vendor will be able to port to new microprocessors in the future.

Initially, all RTOSs were coded in assembly language to ensure that the operating system adds the least amount of overhead possible to the application and that the RTOS can support fast, deterministic response to external events. Porting from one processor architecture to another is not trivial because library routines are specific to a particular software and hardware architecture.

The RISC design discipline is based on the premise that code will be written in a high-level language and that optimizing compilers will generate efficient assembly language programs. As RTOSs became increasingly popular, their vendors were faced with a combination of obstacles in porting to new processors. The most important factor is the language in which the software is written and the most portable language is C. C is the most portable because it is usually the first language provided for a new system. The vendors chose C language implementations for several reasons besides its wide-spread use, including the inherent inefficiencies and unreliable nature of coding RISC processors in assembly language as well as the portability of the code to many different microprocessors.

Beginning in the 1980's, many vendors coded only time-critical scheduling and task switching routines in assembly language, while over ninety-five percent of the executive kernel was developed in the C language. This strategy allows vendors to easily port the C executive to many processor families.



*c. Scalability*

While portability means that the OS has the ability to run on a variety of hosts, scalability refers to how well a system can adapt to support various applications and hardware in the existing system, as well as any future requirements on the system. The language in which an RTOS is written affects how well it can scale to support products. Scalability is particularly important for designers that want to standardize on a single OS with confidence that they will not outgrow the OS.

*d. Multiprocessor Support*

High-end performance-intensive applications may require more than a scalable RTOS environment. In this case, perhaps a single processor is not capable of serving the application. The developers must turn to multiple processors and a software environment that simplifies the software development tasks. Multiprocessor support may therefore influence the choice of RTOS environment.

Ideally, an RTOS would allow development of the multiprocessor environment as if it were a single high-performance processor. It should be possible to develop individual tasks as if for a multitasking environment where the RTOS environment handles task-to-task communications through queues and semaphores, irrespective of whether the tasks execute on the same or different processors.

*e. Extended Services*

Equally important as scalability and multiprocessor support, the number and nature of the extended services offered with an RTOS can determine how well the software accommodates an application. Rudimentary real-time kernels typically only implement a multitasking scheduler, memory management services, an interrupt handler, and communication and synchronization services such as semaphores, mailboxes, and queues. A more robust RTOS environment can include a plethora of I/O and file managers.

*f. Vertical Applications*

In addition to services common to most computer environments, today's RTOSs have a built-in capability to seamlessly function with vertical applications. For example, Simple Network Management Protocol (SNMP) and Motion Picture Experts Group (MPEG) add ons, among others, can be purchased with some RTOSs. These products may be supplied by the original equipment manufacturer, but more commonly, are created by third party vendors in the growing vertical applications market. This is a factor when considering future usefulness of the designed system.

*g. POSIX Compliance*

POSIX is a set of IEEE and ISO standards that define an interface between programs and operating systems. By designing to conform to POSIX, developers have some assurance that their software can be easily ported to POSIX-compliant

operating systems. These systems include most varieties of UNIX as well as Windows NT.

Even with vertical features and diverse capabilities in terms of scalability, it is possible that a single RTOS will not support subsequent projects. When moving from one RTOS to another, standards compliance facilitates the transition. Any RTOS that complies with the POSIX standard would share a standard application-programming interface (API).

POSIX is not an ideal solution for real-time systems as POSIX is derived from requirements for Unix. These requirements include conventions that compromise the real-time goal of minimal latencies. Despite the drawbacks, many vendors attempt to comply with subsets of the overall standard to achieve portability. The two sections of POSIX that apply to real-time systems are section 1003.1 that defines real-time system calls, and section 1003.1B that defines real-time facilities, semaphores, message queues, and signals.

#### *h. Language Support*

Even without true embedded standards, the use of standard languages produces highly portable applications. C is currently the most popular language and it maximizes application portability. Additionally, object-oriented C++ with its modularity enables reuse of existing code in future applications. Despite the popularity and capabilities of C, there are valid reasons to use other languages. For example, the scientific community has millions of lines of existing Fortran code that implement proprietary numerical algorithms. The availability of

languages for a development project is limited to the languages that have been ported to the chosen RTOS environment.

*i. Development Environment*

It is possible to minimize difficulties in moving from one RTOS to another by using a common development environment (vendor-independent from the RTOS) that not only includes language compilers, but also consists of debuggers, editors, profilers, and configuration management and control systems. Despite the advantages of using a development environment from a third party vendor, some developers may find it more advantageous to use a totally integrated environment from a single vendor.

*j. Licensing and Cost*

The final issue for consideration is the cost of the RTOS. The cost and licensing issue decomposes into the development environment and the cost for each run-time license that must be shipped with any product that embeds an RTOS.

Run-time pricing is highly proprietary and is typically negotiated with each RTOS vendor. The price of development licenses is more quantifiable, although prices may be negotiable. Many vendors price their systems on a project basis rather than selling a permanent license to use the software. Whether sold on a project or permanent basis, the licenses are sold per user or seat.

## **B. OBSERVATIONS**

After comparing embedded operating systems offered by different vendors, VxWorks® from Wind River Systems offered all the requirements needed to provide the level of performance required by the TMR system described in this thesis. Also, the numerous features and functionality of VxWorks® will permit the TMR to have the capability to become the basis of future generations of space-based computing systems.

Other candidate embedded operating systems that were evaluated and found to be insufficient to meet the requirements of the TMR system are addressed below.

Specifically, VxWorks®:

- Supports the processor selected for the TMR, the RISC 3081.
- Possesses the capability to be portable to new processors. This is a significant factor to ensure the long-term capability of this TMR concept as new, faster processors come on the market.
- Is very scalable, even across multi-vendor markets, which is important when considering the TMR's usefulness in future projects.
- Provides multiprocessor support.
- Supports a full range of real-time features including fast multitasking, interrupt support and both pre-emptive and round robin scheduling. Its microkernel design minimizes system overhead and responds quickly to external events.

- Supports a robust Java-based GUI development tool that can be used in the HCI.
- In addition to VxWorks®, Wind River Systems also offers its Tornado™ development environment. Tornado™ includes:
  - ❑ An integrated target simulator, which does not require any target hardware or special configuration of the host system.
  - ❑ An integrated version of the logic analyzer for the target simulator.
  - ❑ A project facility.
  - ❑ A debugger engine and GUI.

Other embedded operating systems that were considered, but failed to meet all the requirements were:

- IDT Monitoring System. Packaged with the R3081 processor, it has very limited capabilities which would likely result in a greater development time than VxWorks.
- QNX® RTOS, by QNX, was a very attractive option. Initial research, however, revealed that QNX® RTOS does not support the RISC processors this project is based upon.
- PSOSystem™, by Integrated Systems, was also researched. However, in continuing with the reasoning that this TMR is being built for the long-term we considered it to be important to use the same operating system that The

Naval Research Laboratory (NRL) has used in the past and is planning on employing on future projects. Currently, NPS is collaborating on the TMR with NRL. NRL is also using VxWorks® on several R3000 based processors, namely the RISC 3081 Clementine, and the RH3000 and RISC 3081 USA/Argos. Also, NRL's Solid State Compressive Recorder (SSCR) program is developing quicker processors with greater storage capability. VxWorks® is the RTOS in the RH3000 based multiprocessor that NRL is experimenting with for this program. The TMR is being developed with the intent that it will be used in conjunction with future NRL projects. Utilizing the same embedded operating system will help to ensure seamless integration.

After selection of the operating system, the design, coding, and testing of the HCI as well as the configuration and coding of the operating system were completed concurrently. These topics will be addressed in the following three chapters.

### **III. HUMAN COMPUTER INTERFACE DESIGN**

#### **A. OVERVIEW**

Tornado is an integrated environment for software cross-development. VxWorks is a real-time operating system that runs time-critical or embedded applications. Although the Tornado development environment provides many tools for development and debugging of VxWorks and its applications, upon completion and space deployment, Tornado will be disconnected from the TMR. Therefore, Tornado will not have connectivity to capture register data from the FIFOs after a voter-logic produced interrupt. Also, since the TMR will lack connectivity with the Tornado tools, an alternative was required to set the processor mode once the development and testing process is complete.

To meet the requirements to capture register data and set the TMR processor mode, a human computer interface (HCI) was designed as part of the total system to be ultimately tested in a laboratory environment while undergoing injected faults from a cyclotron. Since the cost of cyclotron testing is very high, it is important that the hardware and software undergoing the fault-tolerance experiment be pre-tested and deemed reliable. In addition, the experimenters need an interface to the TMR system that can assist them to maximize their time while using the cyclotron facility. The primary purpose for the design of our HCI is to provide the user an intuitive interface, with a shallow learning curve to ensure that the majority of their laboratory time will be spent testing their computer product, and not learning a new interface.



The usability of any computer interaction product or application is inherently coupled to the HCI. If the HCI is intuitive, easy to learn and use, the product or application will likely have a favorable usability rating. Guidelines and user-interface heuristics exist on how to best design interfaces for usability. For example, Shneiderman (1997) proposes eight rules of interface design to best maximize the usability of an interface. We selected these accepted rules as the foundation of the design phase, to permit creation of an HCI that inculcates a sense of understandability and competence to users. The eight rules are as follows:

- strive for consistency
- enable frequent users to use shortcuts
- offer informative feedback
- design dialogs to yield closure
- offer error prevention and simple error handling
- permit easy reversal of actions
- support internal locus of control
- reduce short-term memory load

Consistency can be obtained through the use of uniformity in the visual representation of the objects in the interface. Consistency is not always possible to achieve in every instance, but identical symbology and methods of interaction should be employed throughout (Schmorrow, 16).

Shortcuts allow frequent users to reduce the number of interactions required to obtain a desired result and also increase the rate of interaction (Schmorrow, 16).

Offering informative feedback not only helps to reduce frustration on the part of the user, but is also a form of error handling and error prevention. Short, non-meaningful messages such as "syntax error," provide very little useful information and should be avoided. In order to provide useful feedback to the user, it is easy to become verbose which is as bad as providing no feedback at all. Developing meaningful feedback creates more work for the designer, but users appreciate the additional information to assist them in using the system to test their product.

Another feedback design factor is to have the system appear to take blame for errors. An example is the difference between using "illegal command" and "unrecognized command." In the first message, the user is put on the defensive as the user may feel he or she is at fault. In the second example, the error message shifts the blame to the system, yet conveys to the user that the system cannot continue without some new, correct input. The designer must walk a fine balance to create the correct amount of feedback to inform, not overwhelm the user.

Grouping related actions in order to provide a natural flow through the interface can enhance usability of a system. Humans, when learning a new task, will naturally try to order the actions to assist in memorization. By having the interface provide the user with a built-in and easily understood sequence of actions, the user will have an innate familiarity with the outcome.

Whenever possible, a user should be able to reverse actions if they choose. Users make mistakes and they should be allowed to easily recover from any error to reduce their level of anxiety (Schmorrow 18).

The final consideration in the interface design, is the reduction of the short-term memory load on behalf of the user. There is a limit on the amount of short-term information a human can remember. The general rule is seven items, plus or minus two items of information. Therefore, in order to keep the interface as simple as possible, the amount of information the user is expected to remember must be kept to a minimum. Assisting the user in this manner is achieved by the use of cues, mnemonics, and standardized sequences of actions (Schmorrow 19).

## **B. NEEDS ANALYSIS**

Understanding the basic rules helps us to design an interface with a shallow learning curve, low probability of error, and high memorability to permit infrequent users to test the level of fault tolerance afforded by a system. Conducting a needs analysis complements the rules by ensuring that the features and functionality the user would require during testing were represented.

The fault-tolerant TMR and embedded operating system are the foundation upon which the testbed is being built. The testbed provides the designers with tools to test the system in a controlled laboratory environment where a cyclotron will introduce particles to generate Single-event upsets. The HCI has functionality which includes the following:

display various types of information to the user, including number of errors captured and identity of the processor that experienced an upset; save error, register and bit data in secondary storage; load an application; pause and restart the processors from the keyboard; and save a history log of these events.

### **C. USER ANALYSIS**

In order to design the interface for a typical HCI user, we developed a profile of the likely users of the interface. The profile consists of a description of the user based on significant characteristics that may affect the design. Items considered and included in developing the user profile were the following:

1. What will they be using the interface for, and how often will it be used?
2. What is the user's general computer skill level?
3. Is the user familiar with the concepts of fault tolerance and the hazards of operating in the space environment?

The typical user of the TMR testbed can be described as either female or male (over the age of 18), and a professional computer scientist, engineer, or software engineer who is developing software to be used on space-based systems. The user has requisite knowledge of fault tolerance and is thoroughly familiar with the procedures, terminology, and hazards of operating computer systems in the space environment, to include the affects of radiation on integrated circuits. The user is presumed to be well-educated, very comfortable with technology, and very adaptable to change. The user is assumed to be

proficient in English. The typical user has good computer skills and is very comfortable using a mouse and keyboard. The user is knowledgeable and at ease with GUI presentations of information.

The TMR system, in a laboratory environment, will be used during simulated extended periods of radiation exposure on hardware or software. The typical user is expected to spend less than one week per scheduled test. Additionally, due to the high cost involved in these tests, a typical user may experience extended periods between tests, potentially in excess of several months.

#### **D. TASK ANALYSIS**

The product of a task analysis is a hierarchical set of tasks that identifies the functions of the system. In order to develop the listing, the basic tasks that the user should be able to accomplish are identified. "Task analysis involves understanding the required sequences, why they are required, what the information flow is, what the user contributes to the procedure, and what can be automated with the objective of designing a better procedure" (Hix 118). Task analysis is one of the most important aspects of designing effective interfaces. The remainder of this section documents the task analysis performed for this project.

**Primary Task:** Identify errors in fault tolerant hardware and software

Primary Subtask 1: Download test program

Primary Subtask 2: Start program

Subtask 2.A: Start program execution from beginning

Subtask 2.B: Set program runtime

Subtask 2.C: Restart program from beginning

Subtask 2.D: Continue program running from user halted  
position

Subtask 2.E: Display execution confirmation

Primary Subtask 3: View SEU errors

Subtask 3.A: Display error information

Subtask 3.B: Close error window

Primary Subtask 4: View historical log

Subtask 4.A: Display log

Subtask 4.B: Display graphical information

Subtask 4.C: Close historical log

Primary Subtask 5: Save historical log to disk

Subtask 5.A: Select range of data to copy

Primary Subtask 6: Print historical log

Subtask 6.A: Select range to print

Subtask 6.B: Stop print

Primary Subtask 7: Stop program execution

Subtask 7.A: Ability to cancel stop-program-execution  
command

Primary Subtask 8: Pause program

Subtask 8.A: Restart

## **E. CONCEPTUAL DESIGN AND VISUAL MODEL**

Based on the task analysis, a conceptual design and a visual model were created. The purpose of the design and model is to identify key concepts in the HCI to produce a conceptual user-interaction design. The concepts include types of objects, relations between objects, attributes of objects, and actions on the objects, relations and attributes. These are shown in Table 3-1.

Objects	Attributes of Objects	Actions on Objects	Actions on Attributes
Program	a. Name of program b. Language of program c. Source of program d. Destination of program	a. Load program b. Start program c. End program	a. Read name of program b. Find location of program
Data	a. Structure of data b. Origination of data c. Destination of data	a. Save data b. Locate errors in data c. Display data to screen d. Read data from serial port e. Write data to database	N/A
Processor	a. Mode of processor b. Name of processor	a. Start processors b. Stop processors c. Restart processors d. Reset processors	a. Set mode of processor b. Read name of processor
History log	a. Format	a. Print history log b. Save history log c. Write to history log d. Display history log	N/A
Error	N/A	a. Calculate error location b. Display error location to screen c. Write error to history log	N/A
Instruction	Count number of executed instructions	N/A	a. Get instruction count
Timestamp	N/A	a. Get timestamp b. Display timestamp	N/A

Table 3-1. Conceptual Design



Relations between objects:

- Program runs on processors
- Processors execute instructions
- Data results from errors
- Processors experience errors
- Errors generate data
- History log is made from errors

The visual model allowed test subjects, who fit the user profile, to evaluate a simple paper representation of the HCI. Low-fidelity prototypes of the screens were sketched based on the functionality outlined in the task analysis and the function points in the conceptual design.

### **1. Reviewer Analysis**

Two test subjects were selected from the typical user pool who were familiar with the concepts of fault-tolerant hardware and the space environment. Using the low-fidelity prototype and armed with minimal startup instructions, both test subjects were presented with two scenarios. At this point in the research we wanted to examine all possibilities of loading the applications, setting the processor mode and saving the test results, realizing the scope would have to be later narrowed. In these scenarios, the users were asked to use different filenames to save the test results to different locations. Not all the actions they were asked to perform would be implemented in the final design. This allowed observation of their ability to complete the given tasks while measuring the learning curve. Figure 3-1 depicts the process flows for User Scenarios 1 and 2.

User Scenario 1: User loads program from a set of floppy diskettes (5). User desires to test software in both operating modes (all three processors running or a single processor running), with the ability to view the real-time history log on the screen. Upon the completion of each test, the user wishes to write the results to a floppy diskette (using different file names), then print a hard copy. User exits the program using labeled exit button or exit option on file menu.

User Scenario 2: User loads program from a CD-ROM. User desires to test software using only one operating mode (user's option). During program execution, user is told that the on-screen history log is displaying information that suggests to the user that the TMR is not running properly. Based on this information, the User opts to halt execution and decide whether to restart and reload the applications, or resume execution. Upon completion of the test, user wishes to write the results to a zip drive. User exits program using labeled exit button or exit option on file menu.

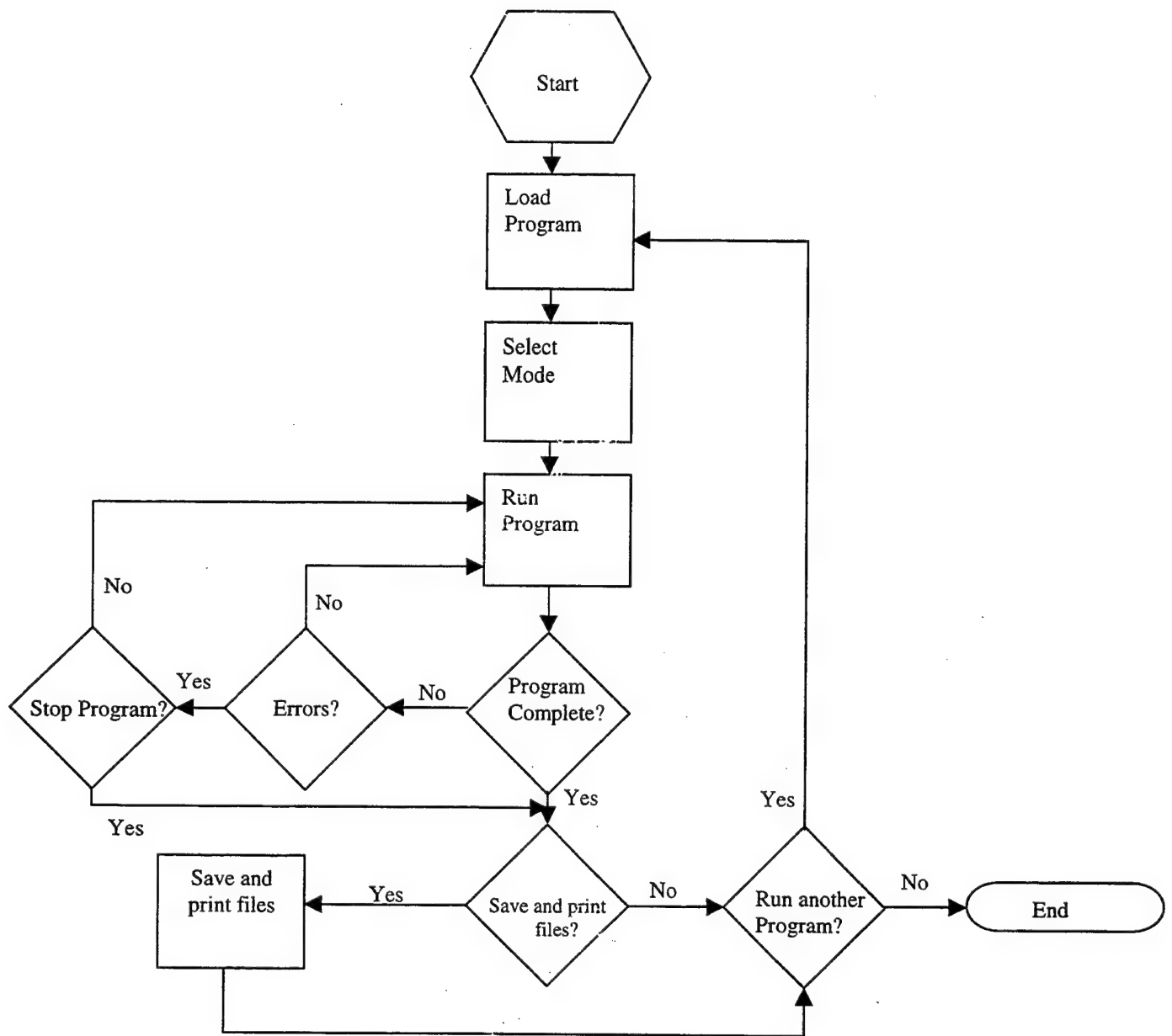


Figure 3-1. Flow Chart for User Scenarios 1 and 2  
(Original Design<sup>1</sup>)

<sup>1</sup> See Figure 4-2 for the flow chart that resulted from the HCI redesign described in Chapter 4.

## **2. Resultant Changes**

Both test subjects performed well during the initial reviewer analysis and provided feedback for improvement. The following changes were implemented in the design as a direct result of the subjects' response to the low-fidelity prototype.

Change 1: Allow the user to print between each program test or after all tests are complete. This will be accomplished by using a drop down box, which lists all files that the user has created. The user can select the file he or she wishes to print. This change was the result of the users desire to have the option to print the test results following each program run.

Change 2: Allow the user to specify not only the file name, but also the location of where to save the data. This will be accomplished by providing the user with a "Save As" dialog box. This change was the result of the users desire to have the option to save to the hardware location of their choice.

Change 3: Allow the user the option of disregarding (not saving) data collected from the most recent program run. This will be accomplished by providing a radio-button "Don't Save" if the user considers the data to be faulty and non-essential.

Change 4: Allow the user to load a test program from multiple diskettes. This will be accomplished by prompting the user to insert diskettes in their logical sequence via a dialog box. This change resulted from the realization that a user may bring their application to the testbed on multiple diskettes, vice only one. As the prototype was originally designed, the use of only one diskette had been considered.

Change 5: In the event of a system crash, inform the user via a dialog box to contact the system administrator. This change was implemented due to the users' desire for timely feedback during application testing. During the reviewer analysis, the users were uncertain about the status of their application (e.g., was it still running, or had the TMR crashed).

### **3. Observation**

During the second test run, a system-generated error was intentionally introduced to observe the user's response. Interestingly enough, both test subjects handled the error in the same manner (by restarting the system) and more importantly, completed the second test run in a shorter amount of time than the first test run. Although this seems counter-intuitive, it is a positive indicator that the interface met the primary design goal, which was a shallow learning curve to ensure the user's time was spent testing their application and not learning a new interface.

## **F. PROTOTYPE**

Following the user feedback from the low-fidelity testing, the next step in the interface design was to create a rapid prototype. The prototyping approach produces an early version of the system to demonstrate features of the final operational system. "With rapid prototyping, the process of constructing prototypes is accelerated, so that the time from beginning a prototype to evaluating user interaction with it is short enough to leave time for substantial changes, if needed, to the product." (Hix, 250) Chapter IV discusses

in detail the iterative process undertaken to turn the low-fidelity prototype into a rapid prototype.

THIS PAGE INTENTIONALLY LEFT BLANK

## **IV. HUMAN COMPUTER INTERFACE RAPID PROTOTYPE DEVELOPMENT AND TESTING**

### **A. PROTOTYPE DEVELOPMENT**

Based on the results of the low-fidelity prototype testing, a rapid prototype for the interface was created as the next step in the HCI design. To complete this phase of the development, it was critical that the resulting prototype would correctly interface with the TMR and display the required user information. Although the successful user-interaction results from the low-fidelity testing indicated the first iteration had been satisfactorily designed to meet the anticipated user interface requirements, the final hardware design was not known, and was therefore not fully considered during the low-fidelity design phase. With the final hardware design completion came a realization that a physical hardware constraint existed that would require significant alterations to the low-fidelity prototype design. This drastic revision became necessary to create the correct interface, despite the unforeseen constraint, to permit correct interactions and correct system operation between the user and the TMR. The constraint and revisions are described below.

#### **1. Rapid Prototype Revision**

The TMR hardware, as designed, is represented by Figure 4-1 and described below.



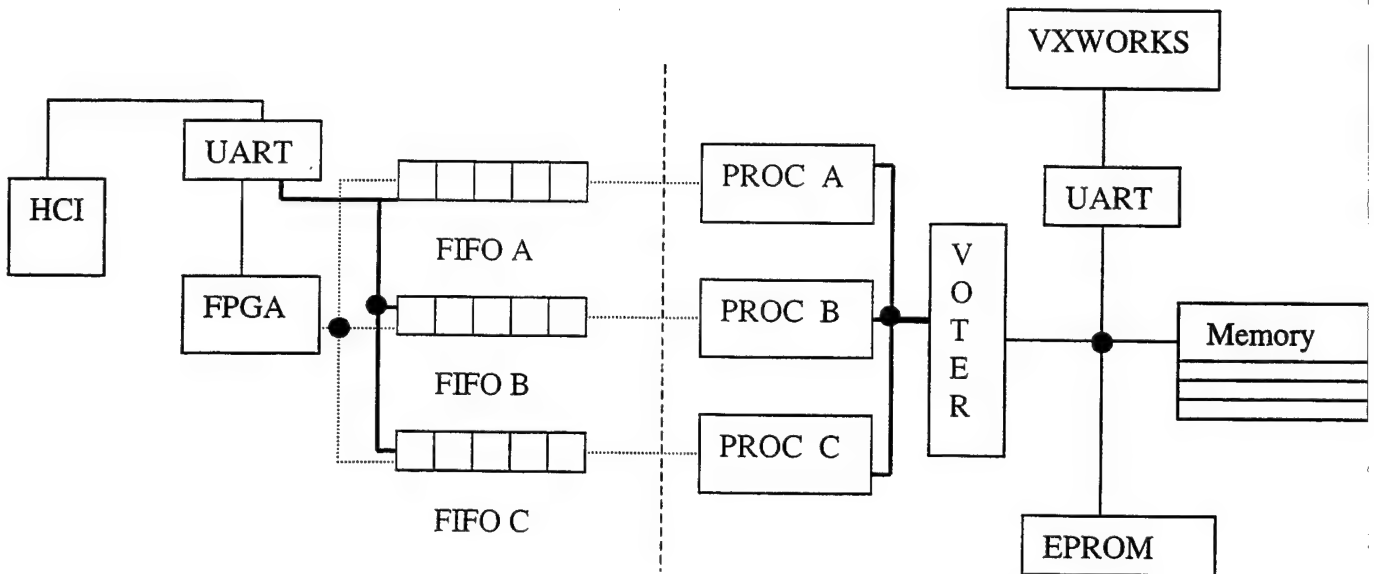


Figure 4-1. TMR Hardware Design

*a. UART*

UART is an acronym for universal asynchronous receiver-transmitter. The UART is a computer component that handles asynchronous serial communication via the serial port.

*b. FPGA*

An acronym for field-programmable gate array. The FPGA is a type of logic chip that is programmed to support the TMR's integrated circuit design. The FPGA supports the system controller that sets the mode of the processors, and transfers data between the TMR and the HCI.

*c. First-In-First-Out (FIFO) registers*

FIFO registers capture the error data from the processors once the voter recognizes an error.

*d. TMR processors*

PROC A, PROC B and PROC C, are connected in such a way that the operating system acts as if there is only one processor in the system. The processors operate in lock step from boot up by executing the same instructions in parallel.

*e. Voter*

The processors are connected to the voter. The voter is comprised of two FPGAs. After each instruction, the voter performs a majority vote on the signals and passes them on to the memory where they are stored. If the voter detects an error, meaning a bit has been flipped, an interrupt is signaled and the FIFO registers begin collecting the processors' data.

*f. EPROM*

EPROM is an acronym for erasable programmable read-only memory. EPROM is a special type of memory that can be reprogrammed. An EPROM differs from a PROM in that a PROM can be written to only once and cannot be erased.

The HCI, as originally conceived required the functionality to:

1. Allow a user to upload an application program onto the processors from the HCI.
2. Set the processor mode.
3. Start the application running on the TMR.
4. Pause the application processes, to allow examination of system state or register contents.
5. Stop the application from running.
6. Reset the processors to repeatedly run the application.
7. Display the errors received for the user to view.
8. Save the errors in a text log, and as actual data received.

However, as seen in diagram 4-1, the HCI's only interface with the TMR is to the UART that is controlled by the system controller FPGA. The HCI has no physical connection to the processors, thereby making it impossible to have any interface to upload the application program onto the processors, or being able to pause or stop the processors. Once this final hardware configuration and the limitations it imposed on the HCI design became known, a reevaluation of the HCI design and its interaction with the TMR was conducted. The purpose of the evaluation was to determine which, if any, of the original eight requirements could be met with a redesigned HCI, instead of having to start a completely new HCI design process.

The re-evaluation results showed that all the functionality requiring interface with the processors, (loading an application program, starting the processors, pausing and stopping the processors) could be performed by the VxWorks interface, see Figure 4-1.

The VxWorks interface, with its physical connection to the processors, along with the Tornado Applications Development Kit, could easily perform the tasks that the original HCI design could not. The newly designed rapid prototype would still have the ability to set the mode, reset the board and the system as needed, find the errors in the transmitted TMR register data, make a log of the errors found, and save the log and data for later examination by the user. Therefore, the redesigned HCI, used in conjunction with the VxWorks interface, could meet all the user requirements<sup>2</sup>. Admittedly, the final HCI rapid prototype was not as elegant as the original design, however the hardware configuration was both the impetus and the limitation for the final design. Figure 4-2 depicts the revised flow chart following the HCI redesign.

---

<sup>2</sup> It was determined that the capability to directly print from the HCI was not a requirement for proper testing and evaluation and that subtask was eliminated from the design.

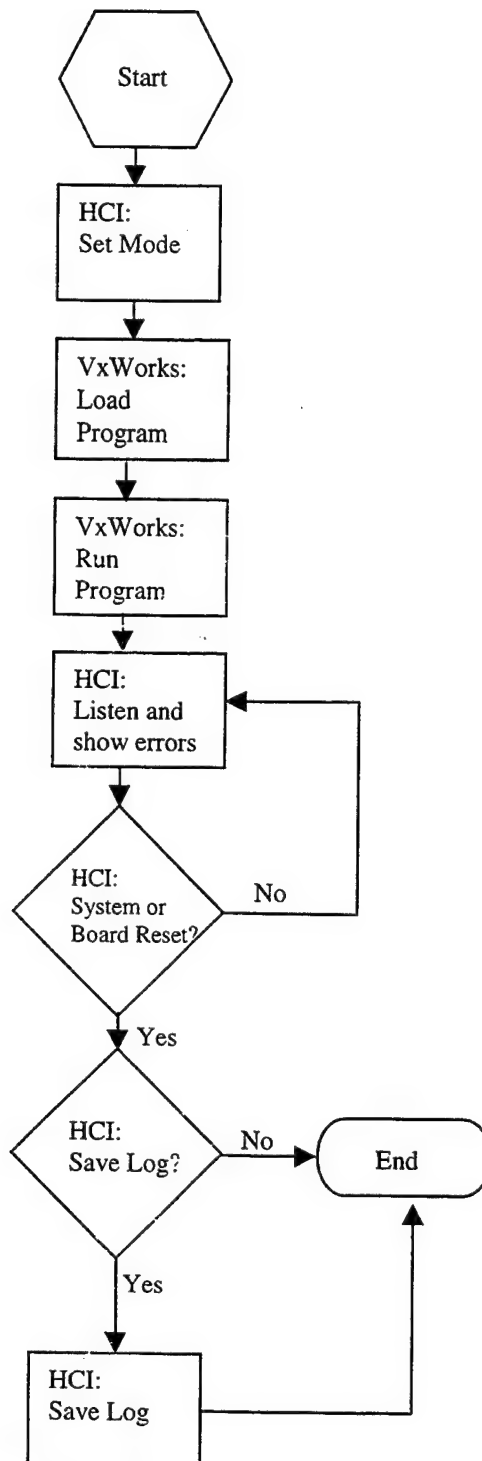


Figure 4-2. Revised Flow Chart

## **2. Rapid Prototype Design**

With the new design, the rapid prototype development began. The first step was to redesign the GUI to eliminate those portions of the original design that VxWorks would perform and were no longer applicable to the HCI. The tasks deleted from the HCI were the ability to upload a program, and pause and stop a program. The next step was to provide additional information on the screen interaction windows to allow the user full comprehension of the testing steps and procedures with respect to the two interfaces. The interaction windows were designed to take into account the fact that the HCI and the VxWorks interface take specific, ordered steps when starting the TMR, and to load and run a program. Given the new design, the user must be able to interact with both of the interfaces (i.e., HCI and VxWorks). The steps that the user must perform are as follows:

1. Select the processor mode from the HCI. The user would select Mode A, which sends a signal to the system controller FPGA via the UART, to run all three processors, or the user would select Mode B, which would indicate that only Processor A, would be run. The processor mode had to be set before the application program was loaded.
2. The user would then load an application program from the VxWorks interface to the processors.
3. The user would execute their program from the VxWorks interface.
4. The HCI would listen for and display error data being sent from the TMR.

5. The user would examine the register contents, system state and application progress from the VxWorks interface.

6. The user would perform system-level resets and board level resets from the HCI.

7. The user would save the history log from the HCI.

To make the steps the user was required to take as intuitive as possible, modal dialog boxes were utilized on the HCI to provide specific information to the user. With the modal boxes, the user had to acknowledge the box prior to taking the next step in the test procedure. The drawback to this method is there was no way of checking to see if the user understood and followed the steps before closing the box, or simply closed the dialog box and ignored the information.

### **3. Modal Dialog Boxes**

Appendix A contains the TMRInterfaceClass.java source code that was written to implement the HCI. The following diagrams portray the modal dialog boxes that were designed for use with the HCI to provide the user with correct information to test the TMR.

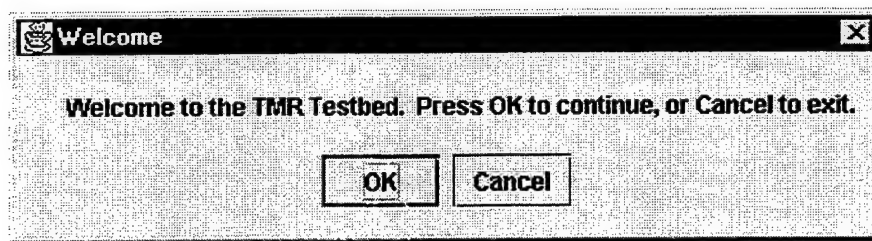


Figure 4-3. "Welcome" Dialog Box

- a. The "Welcome" Dialog box is used to identify the start of the test procedure to the user.

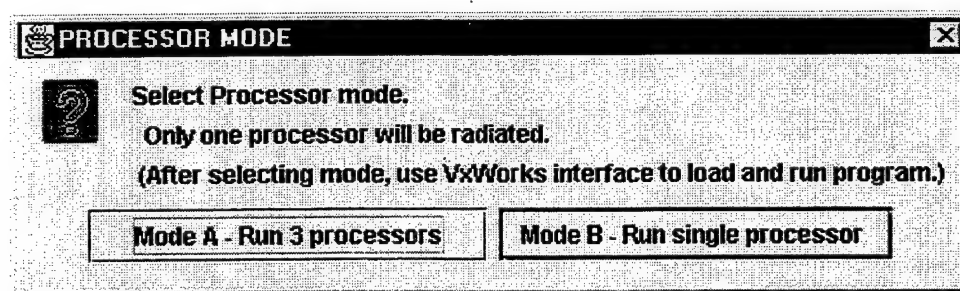


Figure 4-4. "Select Processor Mode" Dialog Box

- b. The "Select Processor Mode" box allows the user to select either Mode A or B; Mode A sends a signal to the TMR that the user desires the application to be loaded and run on all three processors. Should the user select Mode B, the TMR will load the application on one processor only.



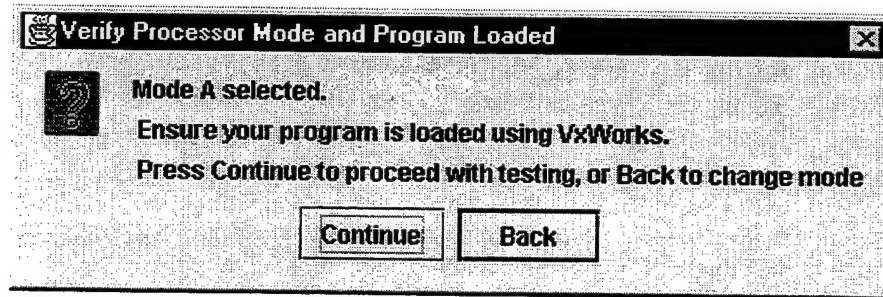


Figure 4-5. "Verify Processor Mode" Dialog Box

c. This dialog box shown in Figure 4-5 allows the user to verify that he or she had selected the mode they desired. If the user had meant to select Mode B, this box gives the user the option to return to the previous dialog box to select Mode B. Also, this box prompts the user to now use the VxWorks interface to load the application program.

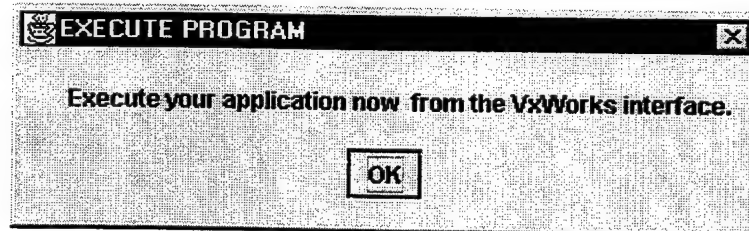


Figure 4-6. "Execute Program" Dialog Box

d. The Execute Program dialog box prompts the user to execute the user's program from the VxWorks interface.

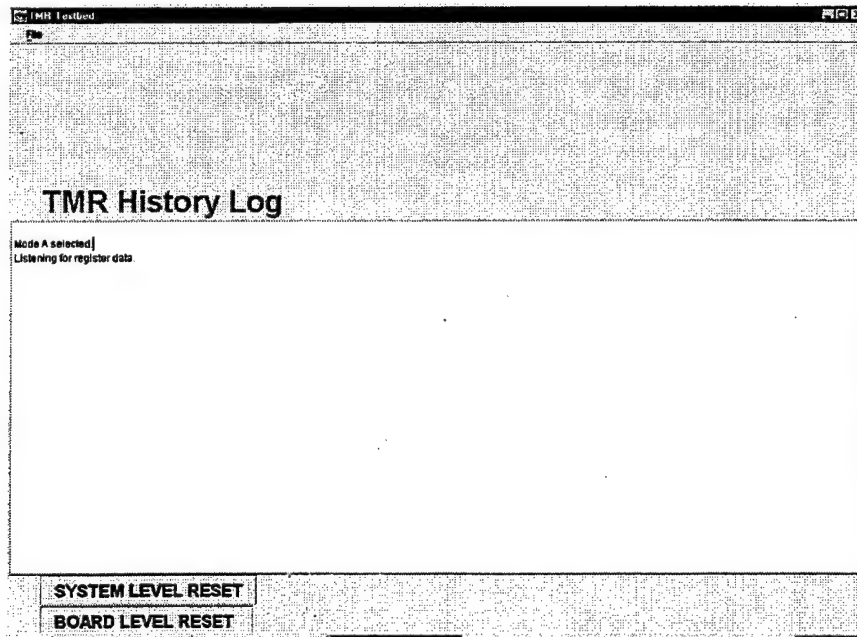


Figure 4-7. "TMR Testbed Main Screen"

e. The TMR Testbed Main Screen is always in the background during the HCI's operation. The user cannot interact with the Main Screen when any of the dialog boxes are present, but can see the contents of the History Log. The History Log is updated with error information as errors occur. The System-Level Reset and Board-Level Reset buttons are available to the user only when no dialog boxes are open. The main screen can be closed, and the HCI program terminated by using the typical window closer, or the File=>Exit menu bar. The window-closer and Exit menu items, when used to terminate the HCI program, also possess a file saving capability to capture the history log.

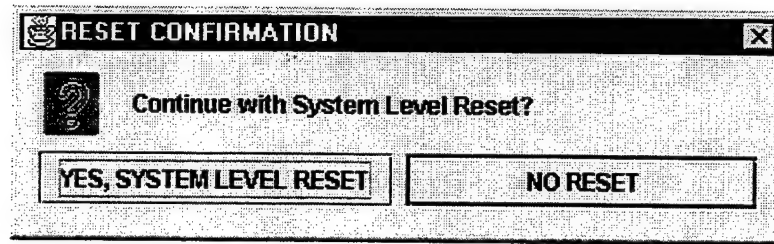


Figure 4-8. "Reset Confirmation" Dialog Box

f. Once the user selects either the System-Level Reset button or the Board-Level Reset button, the Reset Confirmation Dialog Box appears to allow the user to verify whether they desire to reset the system. Should either the system or board be reset, the program will be terminated. Used properly, this reset operation is a desirable feature as it was foreseen that the user would have a need to reset the TMR in some circumstances. However, an inadvertent reset is an undesirable event in the testing process, so the option to select "no reset" is included to permit continuation of the testing process.

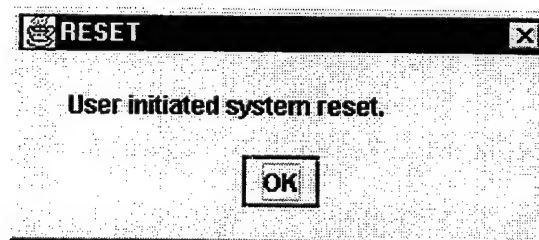


Figure 4-9. "Reset" Dialog Box

g. This dialog box does not allow the user the choice of any options. It is simply to inform the user that he or she had reset the system.

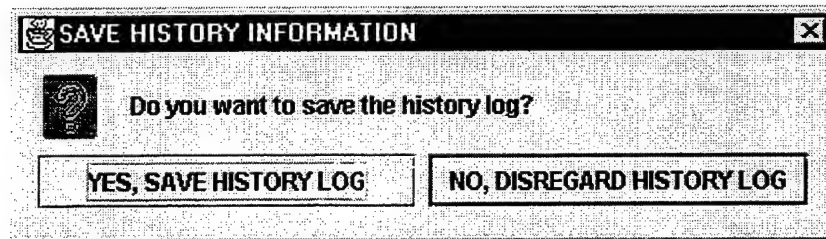


Figure 4-10. "Save History Information" Dialog Box

h. This dialog box allows the user to select either to save the contents of the History Log that is displayed on the TMR Main Screen, or discard it. If the user selects to discard the contents, then the user is given one more opportunity to save before the log is deleted. If the user chooses to save the history log, then the user sees a typical file-saver box. The user would enter the name he or she wants to save the file under, and the HCI would save the file to the E: drive. During program execution, the HCI stores the error data automatically to the database. If the user does not want to save the data, it can be accessed and deleted.

A desirable feature in the final HCI product would be to allow the user the option to run another test on the TMR after they have saved the History Log. This could be accomplished by bringing the user back to the Select Processor Mode Dialog Box (Figure 4-4) where they would follow the same steps as they previously completed to run another test. This functionality was not implemented

in the rapid prototype. The rapid prototype exits the HCI program after the user saves the History Log

#### **4. Error Detection**

Upon completion of the HCI rapid prototype, the next step was to develop effective communications between the TMR and the HCI. Because the hardware utilizes serial port communications and the HCI was written in Java, the logical choice for the HCI's communication capability was the JAVAX Communications Package. The JAVAX Communications Package contains all of the necessary interfaces to design a serial port for the HCI. Using the predefined interfaces, vice creating a new communications protocol, was deemed to be the best option because the JAVAX Communications Package has been thoroughly tested and is well documented. Additionally, there were simple examples available to provide a foundation to begin building the communications interface for the HCI. Appendix B contains the ReaderThread source code that was written to implement the communications functionality for the HCI. ReaderThread was implemented as a thread.

Once running, the HCI creates a serial port connection using the ReaderThread file. The ReaderThread created a listener to listen for incoming TMR error data, stores the data into a queue, (Appendix D, FifoBuffer.java), and signals the program when data has been received on the port.

After the communications interface was successfully designed and compiled, an algorithm was developed to correctly identify the error (bit flip) that initiated the TMR dump of the register contents. The solution requires an algorithm capable of comparing

the contents of the three processors to determine which bit has experienced the error. The final solution is as follows:

1. The voter in the hardware examines the register contents after every cycle. If the contents of the three processors do not exactly match, the HCI receives the contents of those registers via the serial port.

2. Once error data is received on the serial port, the serial port listener awakens and takes all of the error data, one byte at a time, and puts it into a synchronized queue. After all of the register data from one error has been received and entered into the queue, the serial port listener turns the error detection over to an error-detecting thread. The listener begins listening for more error data.

3. The error detecting thread takes the error data out of the queue in the order it was received and using nested for loops, puts the error data into three arrays, one array for each processor's error data. Once the error data is in the arrays, the arrays are compared using the "exclusive or" (^) operator. In Java syntax, the "exclusive or" is represented by "^". The "exclusive or" operator returns a "zero" only if both sides of the comparison are the same. Any difference in the two arrays would have resulted in a return other than zero. Once the non-zero return is detected, indicating a mismatch, then it has to be determined which of the two arrays the error had occurred in. This is conducted using a process of elimination between the comparisons. The logic used is shown in Figure 4.11.

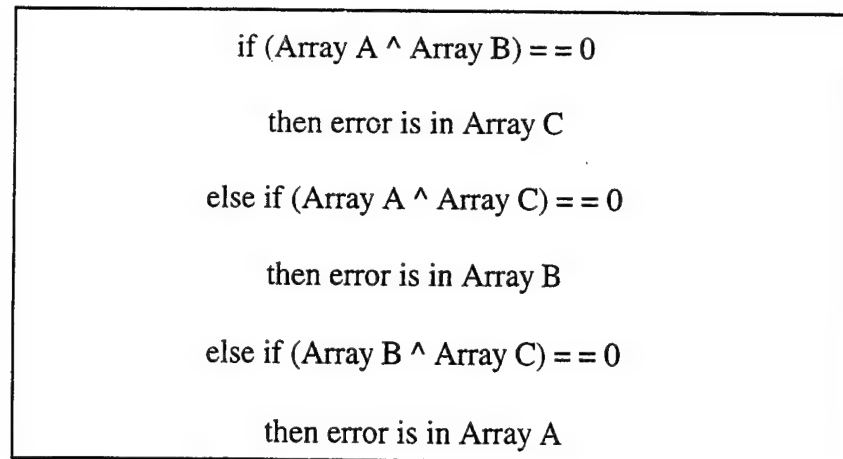


Figure 4.11. Array Comparison Logic to Determine Error Processor.

4. To locate the exact bit that had been flipped, a bitwise comparison between the array that contained the error, and an array that contained no errors was conducted. Each bit was examined using a right bit shift operation. (Appendix C. buildDataThread.java, and the findFlip() method, contain the actual code.) In this manner, the specific bit was found.

5. After the bit is detected, an entry is made to the log on the user's screen which indicates the time, the processor, register and bit that experienced the error. Additionally, the error processor, register and bit are entered into an Access database table, TMR Data, and the register contents from each error were entered into another database table, Registers. A key, comprised of time and processor, ties the two databases together to permit queries.

## 5. HCI Testing

To verify that the designed algorithm did properly identify the flipped bit, a test program was created that generates simulated register data that contains a flipped bit. The test program allows the tester to utilize a very simple GUI to signal to the program to simulate an error from either processor A, B, or C. The test program then runs the algorithm (Appendix E, TMRTestPanel.java) that randomly selects a processor, register, and bit to represent the error. The test program generates a stream of data that contains the simulated error data from one processor, and simulated error-free data from the other two processors. The stream layout is shown in Figure 4-12.

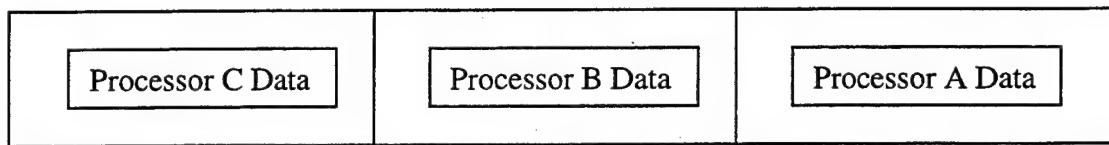


Figure 4-12a. High Level Example of Test Data Stream After Error Detected

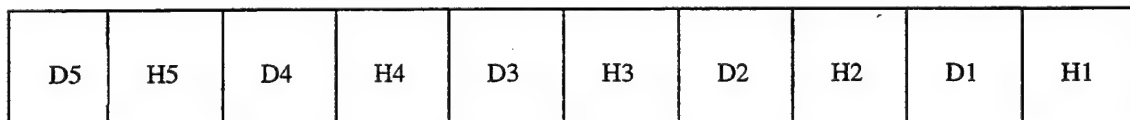


Figure 4-12b. Individual Processor's Data Stream

Figure 4-12a shows the order in which the processors' error data is transmitted to the HCI. Processor A was first, followed by Processor B, and then Processor C. Figure 4-12b show how the data stream for each individual processor is arranged. There is a header (H1 – H5) before the contents of each register (D1-D5). The 8-bit headers are



generated by the hardware and are designed to contain specific information about the data that follows. See Figure 4-13 for a description of the header format.

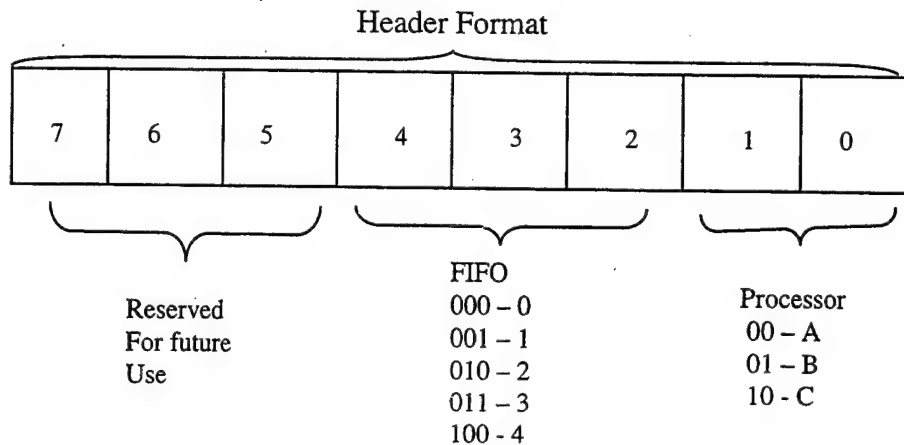


Figure 4-13. Header Format

The first two bits of the header identify the originating processor. The next three bits identify the FIFO register, and the last three bits are reserved for future utilization.

The data blocks D1 - D5 contain the actual information from the registers. Even though each FIFO register has the capacity to hold 4096 addresses, the TMR initially utilizes only 82<sup>3</sup>. This means that each FIFO register holds forty-one addresses and forty-one items of data. Four of the five FIFO registers each hold 8 bits of the 32 bits of register information. The remaining FIFO, FIFO 4, collects control information. D1 is

---

<sup>3</sup> The 82-address capacity was an initial estimate based on known minimum number of registers (not including the floating point registers) transferred during interrupt handling. The final number of registers to be saved will increase when all registers are included. A final number can be determined upon completion of future research concerning the determinism of the operating system and its handling of interrupts.

the most significant byte, D2 is the next most significant byte, down to D4 which is the least significant byte of the 32 bits. Figure 4-14 shows the FIFO registers format.

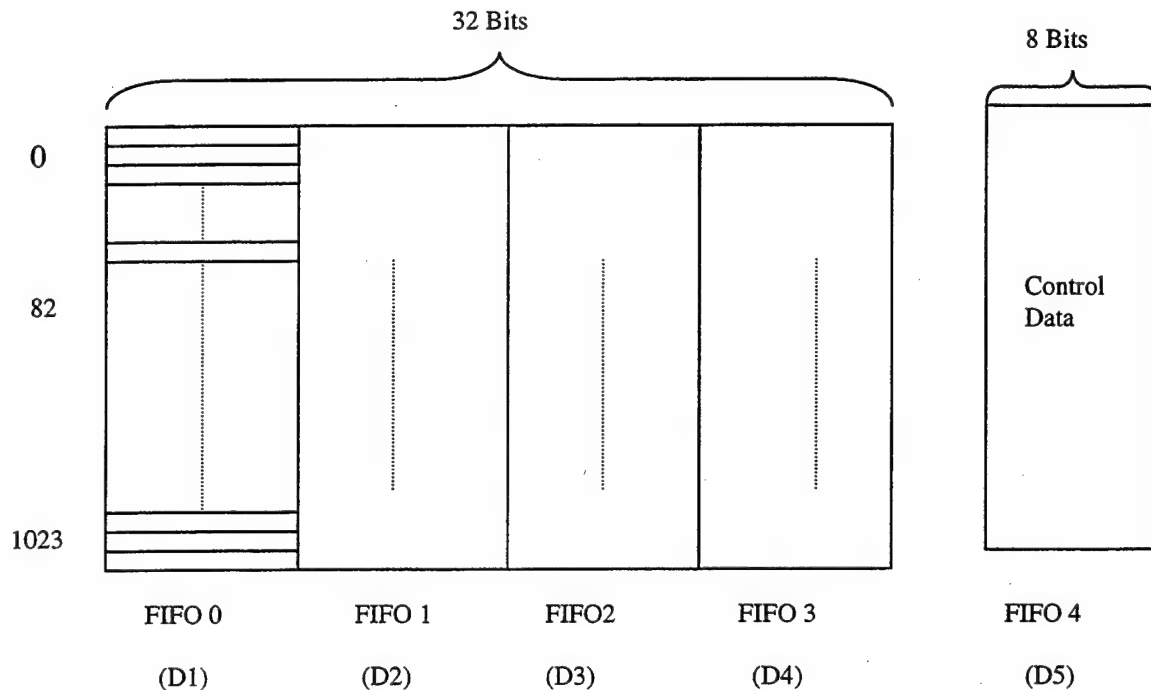


Figure 4-14. FIFO Register Format for Each Processor

The test program generates a stream of data that represents the arrangement of the TMR registers, as well as how the TMR transmits the data to the HCI. The TMR test program also displays which bit it has randomly flipped to assist in development and troubleshooting of the HCI.

To best test the HCI's error detection capability, the simulated data the HCI received from the test program had to be as close in length and format to the actual TMR data stream as possible. Also, the simulated data had to be generated at completely

random times, with completely random bits being flipped to represent an error. This was accomplished by doing the following:

1. Eliminating the GUI from the test program that allowed the tester the ability to generate an error at a specific time and from a specific processor. Instead, a random number generator was put into place to select random times, and a random processor. Since the test program already generated the random register and bit, that was not altered.
2. Seven tests were run using the test program with the HCI. Tests one through six generated 35 errors at random times, with the time interval between errors received by the HCI being between two and seven seconds. The HCI correctly detected every error, logged it to the screen and stored the data in the Access database. Test 7 generated random errors for over one hour to determine if the HCI could withstand a longer test. After one hour, the HCI was still responding to the error data transmitted from the TMR and detecting the bit flips, logging the entries, and storing the data in the Access database. No ability to test the accuracy of the errors discovered against the over 600 generated errors was available.
3. Two tests were conducted to determine whether the test program received the system-level reset and board-level reset signals. In both instances, when the test program received the signal to reset, data transmission was halted, signifying that the test program correctly identified the reset signals.

## **B. DISCUSSION**

### **1. Test Results**

The error-free results of the testing between the test program and HCI indicate that the HCI rapid prototype, as designed, could support the TMR in a laboratory environment, with the TMR undergoing radiation testing. The HCI correctly identified the flipped bit in the error data, and was able to correctly signal the test program to perform a reset. The HCI correctly logged the error information and correctly saved the register contents into a database for later analysis.

### **2. Value of Storing Register Contents**

The information being stored in the Access database could have many useful applications. Software developers, in the interest of creating more resilient software, would be keenly interested in knowing if a particular piece of hardware consistently had an error in a specific bit. This way, that troublesome register could be avoided. Another potential application would be for designers of fault tolerant software who utilized the TMR testbed to test the capability of their software in a radiation environment. Their interest in the data would be in determining if their software caught and/or corrected all the errors, rather than the fault tolerant TMR performing that task.

Examples of the types of information that could be ascertained from queries on the Access database are which register or bit had the most errors, what percentage of the errors did each processor experience, what is the distribution of errors across the registers.

The register contents in the Access database table Registers, are stored as integers. Access possesses the capability to convert the integer to its bit representation should the user need to display the data in bit format.

Once the HCI rapid prototype design and testing were complete and the hardware design was finalized, the next step was to create a Board Support Package for the TMR system. The details of this process are described in Chapter V.

## V. BOARD SUPPORT PACKAGE

### A. BACKGROUND

In order to prepare the embedded OS (VxWorks) for the target hardware (the TMR), the Tornado development tools were utilized. These tools included the following:

Launch	-Launch Tornado Tools
WindSh	-Access target interactively
CrossWind	-Source-level debugger
Browser	-Display system information
Project Facility	-Configure applications or VxWorks
WindView	-Analyze multitasking application
Simulator	-Simulate VxWorks target on host OS

The coding and building of the applications are completed on the host using the Tornado tools. This includes editing and compiling/assembling/linking either within the project facility or from the command line. The testing and debugging is done on the target utilizing the target host tools, including loading, execution, source-level debugging, and performance monitoring. The development cycle consists of iteratively writing and compiling code on the host, downloading to target, testing code on the target, and going back to the host for further writing and modification of the code.

The Tornado environment provides a full range of features. Tornado facilities execute primarily on a host system, with shared access to a host-based dynamic linker and symbol table for a remote target system. Figure 5-1 illustrates the relationships between the principal interactive host components of Tornado and the target system. Communication between the host tools and VxWorks is mediated by the target server and target agent (Figure 5-1 Wind River Systems, Tornado 2).

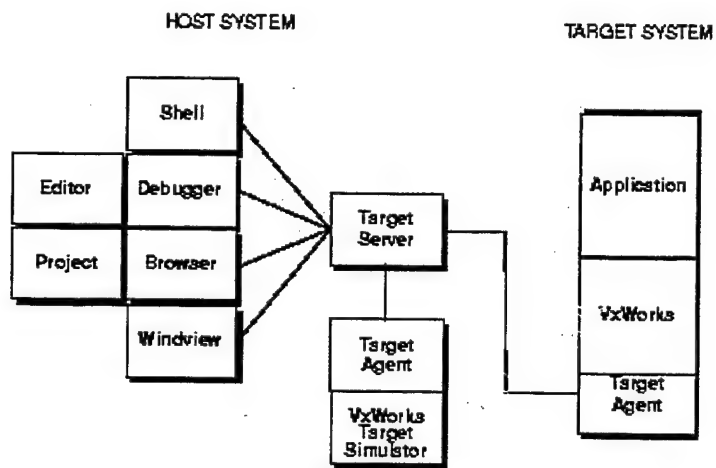


Figure 5-1. Host Tools Communication

The target-host interface is composed of the following three elements: the target agent, the target server, and the target registry which provides the link between the target and host environments.

- The target agent is a scalable component of VxWorks that communicates with the target server on the host system.

- The target server connects Tornado tools such as the shell and debugger with the target agent.
- The Tornado registry provides access to target servers, and may run on any host on a network.

For this research, the target application is a simple program. VxWorks Tools and applications are hardware independent. The tie that connects the application and tools with the hardware is the Board Support Package (BSP). The BSP consists primarily of the hardware-specific VxWorks code for a particular target board. A BSP includes facilities for hardware initialization, interrupt handling and generation, hardware clock and timer management, mapping of local and bus memory space, and so on. BSPs also include project files that facilitate creation of projects for bootable applications and custom VxWorks images. Creating the BSP and writing the application code for the operating system is the focus of this chapter.

## **B. CREATING A BSP**

A BSP consists of the routines that provide VxWorks with its main interface to the hardware environment. Figure 5-2 (Wind River Systems, BSP 12) illustrates the components indicating the hardware-dependent and hardware-independent elements.



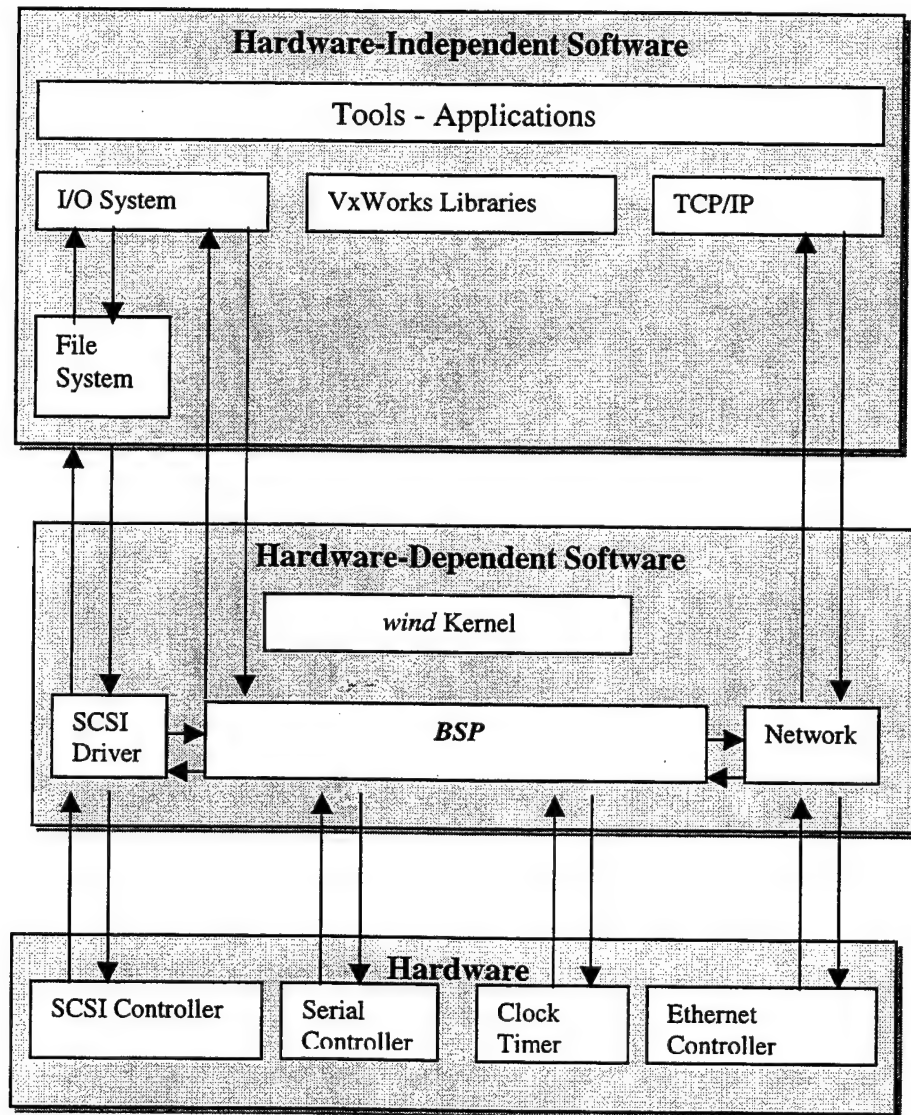


Figure 5-2. Hardware Dependent and Independent Software

Creating a BSP is best completed in a series of graduated steps, each building on the previous step. The steps are as follows:

1. Set up the basis of the development

2. Write the BSP pre-kernel initialization code
3. Start a minimal VxWorks kernel and add the basic drivers for timers, serial devices, and an interrupt controller
4. Start the target agent and connect the Tornado development tools
5. Complete the BSP and include bootROMs, caches, etc.
6. Generate a default project for use with the new project facility

The following sections provide a detailed description of each of the steps listed above.

### **1. Basis of Development**

The VxWorks/Tornado package we purchased was bundled with an idts381 BSP. This BSP was a suitable starting point to create the BSP for the TMR system because it supports the same processor. The reference BSP contained the code required for the processor. However, the serial I/O chip and timers are different. Starting with the basic code required for the processor and then adding board specific drivers reduced our development time.

We were limited when selecting the technique for downloading the code to the target. Since we do not have a ROM emulator or an in-circuit emulator (ICE), we are required to use the download protocol supplied in the board vendor's debug ROMs.

Popular methods for testing the downloaded code include using an ICE, a logic analyzer, or the board's native debug ROMs (given that they support breakpoints). Since none of these methods are available, future work will require use of a debugging library (flash an LED or transmit a character over a serial line in polled mode).

There are three choices for VxWorks image types and the details of the pre-kernel initialization depend on VxWorks image type characteristics. They are as follows:

- ROMable image – Boot or “end-user” image contains bootstrap code which copies VxWorks from ROM to RAM. The ROMable image can be either compressed or uncompressed.
- ROM-resident image – Boot or “end-user” image which executes in ROM and only copies the data segment into RAM. The image starts faster and uses less RAM than the other ROMable images, but executes more slowly because the text (executable) segment remains in ROM and therefore is limited by data widths and lower memory access time of the EPROM.
- Downloadable image – “end-user” image that does not contain the bootstrap code to copy itself out of ROM into RAM. A downloadable image requires a separate program to obtain the image and load it into RAM. It cannot be built using the project facility and must be configured and built using a BSP mechanism.

The TMR contains adequate ROM (512KB) and RAM (1MB). Therefore, we were not constrained by the RAM and could take advantage of the speed gained by copying the entire image into RAM. For this reason, an uncompressed ROMable image is the best choice for the TMR.

## 2. BSP Pre-kernel Initialization Code

The power-up bootstrap code consists of **romInit()** and **romStart()**. The bootstrap code executes the following:

- The processor is forced to the starting address for **romInit()** in ROM
- **romInit()** resets the processor, initializes memory and performs all other hardware initialization.
- **romInit()** then branches to **romStart()** which loads the ROM image into RAM

The primary responsibility of the pre-kernel initialization is to place the hardware in a quiet state so that the kernel can be activated. The pre-kernel initialization sequence is described in Figure 5-3.

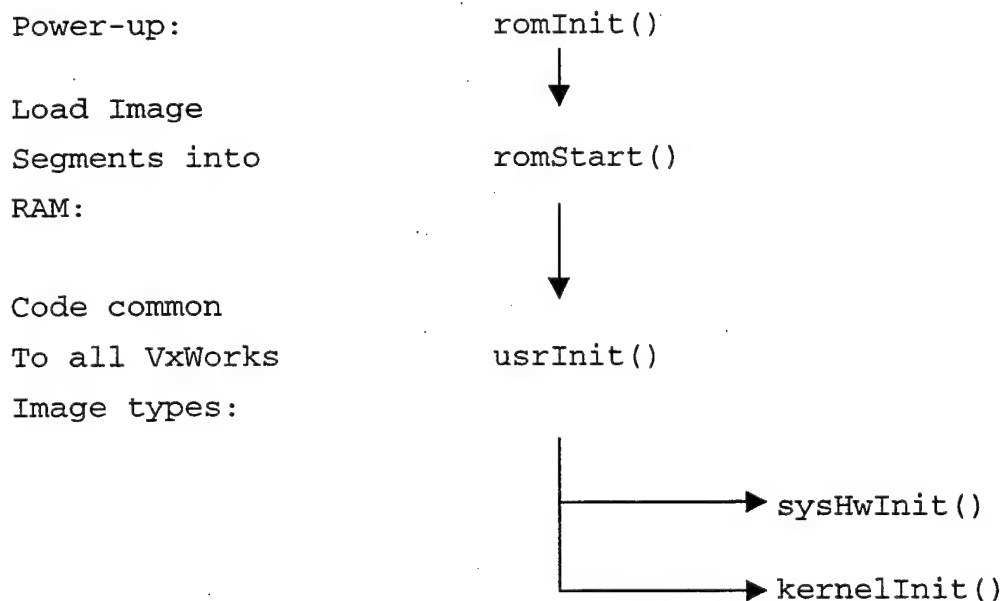


Figure 5-3. Pre-kernel Initialization Sequence

The **romInit()** and **sysHwInit()** are BSP files that were written to reflect hardware-specific features. The **romStart()**, **usrInit()** and **kernelInit()** functions are generic routines.

The **Makefile** was modified from the IDT version. The following macros are required:

- **TGT\_DIR**

The path to the target directory

- **TARGET\_DIR**

The BSP directory name

- **VENDOR**

The board manufacturer's name

- **BOARD**

The board name

- **ROM\_TEXT\_ADRS**

The boot ROM entry address in hexadecimal

- **ROM-WARM-ADRS**

The boot ROM warm entry address in hexadecimal

- **ROM\_SIZE**

The ROM area's size in hexadecimal

- **RAM\_LOW\_ADRS**

The address where VxWorks (the complete, linked VxWorks binary to be run on the target) will be loaded

- **RAM\_HIGH\_ADRS**

The destination address in RAM when the boot image is copied from ROM to RAM; this is the load point for the boot program for a downloadable image as well as the start of the text segment. The boot program will eventually be overwritten

- **HEX\_FLAGS**

Architecture specific flags that will be the same as the reference BSP

- **MACH\_EXTRA**

Any machine-dependent files

Among these macros, the only significant change from the IDT board was the **RAM\_HIGH\_ADRS** macro. This macro identifies the destination address used when copying the boot ROM image to RAM.

Since the TMR board does not include a sonic Ethernet chip, the **config.h** file was modified to undefine **INCLUDE\_SN** and **SONIC\_CONTENTION**. Since the TMR does not support non-volatile RAM (NVRAM), the correct bootline must be burned into the boot ROMs or typed after every reset/power-up during development (Wind River Systems, BSP 67). Additionally several macros dealing with NVRAM had to be set to reflect the absence of NVRAM. The **LOCAL\_MEM\_SIZE** macro had to be changed to reflect the 1MB of memory available in the TMR. According to Wind River Systems, a common mistake of BSP writers is failure to realize that **LOCAL\_MEM\_LOCAL\_ADRS** is not zero; it has to be offset by the start of memory (Wind River systems, BSP 37). The

LOCAL\_MEM\_LOCAL\_ADRS (in this case, 0x800f80000) was included with the appropriate address in the **config.h** file. The RAM\_HIGH\_ADRS macro had to be changed to be consistent with the **Makefile**.

The **sysLib.c** file is the largest file and was implemented in phases. The required functions in the pre-kernel initialization phase are as follows: **sysModel()**, **sysBspRev()**, **sysHwInit()**, **sysHwInit2()**, and **sysMemTop()**.

**romInit.s** had to undergo significant changes. The IDT board contained significant DRAM configuration code that had to be removed. At power-up the processor begins execution at **romInit()**, which must be the first routine in the text of **romInit.s**. For warm reboots, the processor routine begins at **romInit()** plus an offset. Most of the hardware initialization begins in the **sysHwInit()** located in **sysLib.c**. It was important not to try to accomplish too much device initialization in the **romInit.s** (Wind River Systems, BSP 37).

**sysALib.s** contains the entry point for RAM-based images. The entry point, **sysInit()**, performs the required minimal setup to transfer control to **usrInit()**. **sysInit()** generally masks processor interrupts and sets the stack pointer. It is similar for all architectures and was not modified for the TMR.

**romStart()** located in **bootInit.c** contains the code to copy the text and data segments from ROM and into RAM using **vxWorks\_rom**. **romStart()** is generic for all hardware platforms and should not be modified.

During implementation, we wanted to add debugging code to the generic (non-BSP) configuration files. However, instead of modifying the generic files, a copy of the files

was moved to the BSP directory and modified. Specifically, **config/all/usrConfig.c** and **config/all/bootinit.c** were moved to the **tmr3081** directory. The **usrConfig.c** was modified to remove everything except the pre-initialization code, and the body of **usrRoot()** is empty. The following lines were added to the **Makefile** after the definition of the **HEX\_FLAGS**:

```
BOOTINIT = bootInit.c
```

```
USRCONFIG = usrConfig.c
```

### 3. Start a Minimal VxWorks Kernel and Add the Basic Drivers

A minimal kernel involves adding a few device drivers to the kernel initialization. The only driver required by VxWorks is the timer (Wind River Systems, BSP 54). The timer for the TMR will be programmed in the field programmable gate array (FPGA). A driver was written for this timer and placed in the **tmr3081** subdirectory. It is important to place any custom drivers in the BSP directory itself and not the driver directory. This prevents the custom drivers from being erased should it become necessary to reinstall VxWorks. The board-specific initialization was performed in **sysHwInit()** and connected to the interrupt by calling **intConnect()** in **sysHwInit2()**.

Wind River Systems supplied the **ns16550** driver for the Serial Input/Output driver. SIO chip board specific initialization was primarily accomplished in **SysSerial.c**.

The fundamental problem for the TMR BSP was the requirement to determine what was on the address bus during an interrupt initiated by the voter logic. Once the interrupt was received, the FIFO's begin sniffing the bus and dumping all information to the HCI. It was imperative to find the VxWorks code for an interrupt to capture the



information that is provided to the HCI via the FIFO's. Additionally, upon interrupt, VxWorks saves only a subset of the registers.

Initially, consideration was given to writing an Interrupt Service Routine (ISR) within the VxWorks application. A prototype was developed that saved the contents of all the registers upon interrupt. On closer evaluation, it became clear that this approach would send incorrect register contents to the HCI. This occurs because the VxWorks interrupt initiates the saving of a subset of registers. The process of saving the registers moves register contents and therefore corrupts the desired output to the HCI of register information at the time of interrupt. A C command at this point would provide the HCI with register contents very different from the contents of the registers at interrupt. Clearly the best solution would be to examine the VxWorks source code and provide the HCI with a mapping of register data on the bus as a result of VxWorks saving the register subset. In order to capture the entire contents of all the registers upon interrupt, an Interrupt Service Routine must be written to acknowledge the interrupt and provide the additional register information, including the floating point registers to the HCI. An ISR must then be initiated with an `intConnect()` command. The ISR is the subject of future work and is discussed briefly in Chapter VI.

#### **4. Start the Target Agent and Connect the Tornado Development Tools**

The debug agent within Tornado is initialized by a call to `wdbConfig()`. This call is made at the very end of `usrRoot()`.

## **5. Complete the BSP**

Very little additional work is required to complete the BSP. The TMR is a relatively simple board without NVRAM, network devices, Memory Management Units, Dynamic Memory Access configurations requirements.

## **6. Generate a Default Project for the New Project Facility**

A *project* contains the source code files, build settings, and binaries that are used to create a downloadable application, a custom VxWorks image, or a bootable application. A *downloadable application* consists of one or more relocateable object modules, which can be downloaded and dynamically linked to VxWorks, and then started from the shell or debugger. A *bootable application* consists of an application linked to a VxWorks image. The image can be configured to include or exclude components of the OS, as well as resetting the operating parameters. A *workspace* contains one or more projects (Wind River Systems, Tornado 72). The actual creation of the project took place within the Tornado environment. The BSP is compiled and built within the project facility. The project facility provides graphical, automated mechanisms for creating applications that can be downloaded to VxWorks, for configuring VxWorks with selected features, and for creating applications that can be linked with a VxWorks image and started when the target system boots (Wind River systems, Tornado 71). Appendix F discusses generation of a new project based on a BSP.

## **C. DISCUSSION**

During the writing of the BSP, certain architecture considerations dictated the requirements for the BSP and are briefly discussed here. The architecture design of the CPU affects how it is used within the VxWorks system. The MIPS processors can run in either big-endian or little-endian mode. However, VxWorks only supports MIPS in the big-endian mode of operation. Although the MIPS processors include a minimal memory management unit called a Transition Lookaside Buffer (TLB), VxWorks does not support the use of a TLB. Additionally, the MIPS processors support three modes of operation: user mode, kernel mode, and supervisor mode. The VxWorks kernel operates in kernel mode at all times.

## **D. LESSONS LEARNED**

Many lessons were learned during this thesis with respect to the embedded operating system. Initially, we assumed that since VxWorks supported the RISC 3081 processor, a BSP would already exist to support the TMR board. The BSP however is specifically written not only for the processor but also for the board itself including the board specific chips (ethernet, timer, and serial input-output). The requirement to write a BSP was not considered during the initial assessment and turned out to be the most significant effort for the operating system.

Additionally, we did not initially believe the requirement existed to obtain the VxWorks source code. However, to accurately predict the information gathered by the FIFO's, it became necessary to acquire the source code from VxWorks. Clearly, had we

written our own operating system, this would not have been an issue since we would have programmed the saving of the registers ourselves and would have controlled the order of information on the address bus.

THIS PAGE INTENTIONALLY LEFT BLANK

## **VI. CONCLUSION AND FUTURE DIRECTIONS**

### **A. CONCLUSION**

The overall goal of this research was to explore the use of a COTS operating system in conjunction with a fault-tolerant TMR COTS microprocessor in order to realize a system that is able to withstand the rigors of operation in a space environment, in particular, to detect and recover from single-event upsets. In addition, the research involved the development of an HCI that provides both a graphical user interface display and information storage mechanism.

The TMR operating system was configured and the HCI was designed primarily for a ground-based operational testing of the voting logic and any software running on the processors themselves. Currently, the operating system is configured so that an application can be downloaded using Wind River System's Tornado Tools and run on the TMR. The HCI supports both near real-time error detection and post-testing analysis of the state of the three microprocessors. The TMR provides the capability to analyze the success or failure of attempts to improve the performance of COTS microprocessors in the space environment.

The original TMR testbed design was not intended to be used for space flight. However, this research has moved the testbed one step closer to improving and preparing the TMR design for operational use in fault-tolerant space-based microprocessors.

## **B. FUTURE DIRECTIONS**

To further this research, an analysis must be conducted of the VxWorks source code to determine the order the register data is saved when an interrupt occurs. If all the desired registers are not saved, an ISR can be written to save any additional registers required for analysis. When the VxWorks code is analyzed and, if required, ISR code is written, the order of the data on the address bus can be correlated to the entries in the HCI database. The HCI must rely on a predetermined order to permit the error-detection algorithm to correctly identify the location of the flipped bit. Knowledge of the order of the data is essential because that is the only way the HCI can identify where the error originated. Depending on the outcome of the analysis of the VxWorks source code and the order the register data is saved, the error-detecting algorithm may have to be altered.

After hardware testing and evaluation is complete, additional research includes separately testing the operating system and the HCI rapid prototype with the actual TMR hardware. Finally, complete system testing of the TMR hardware, operating system, and HCI rapid prototype must be conducted. The system can then be scheduled for radiation testing using a cyclotron. Preparation for eventual space flight includes research to determine the optimal configuration of the operating system as well as burning the operating system and desired application onto ROM.

Future work for the HCI includes optimizing the database storage code, and research and development of desired query capabilities and a database interface to provide information about the state of the processors. One of the benefits of storing the data into an Access database is to provide the user with the ability to perform ad-hoc queries on the database. However, an interface containing commonly used queries can be designed for ease of use. The ability to open the Access database and perform queries on demand while the program is running is necessary to permit real-time analysis of the performance of the user's application during testing. Research into the storage and retrieval capabilities and connectivity required for a long-term space flight is needed to determine the hardware and processing requirements for large amounts of data collected during these types of missions.

The use of two separate interfaces for testing purposes is not a desirable implementation for use by the experimenters. Research to combine both the Tornado Tools and the HCI onto a single laptop would be beneficial. With the current hardware design, a PC or laptop could be configured with two serial ports; one line connecting the VxWorks interface, and the other the HCI interface, to the respective serial ports on the TMR board. Both the HCI and VxWorks user interfaces could be displayed on a split screen. The user would be able to select the desired interface by gaining focus with a mouse click. Neither the operating system nor the GUI should require significant configuration changes to implement this design.



THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX A. TMRINTERFACECLASS.JAVA

```
//TMRInterfaceClass.java
//A class that instantiates a GUI for the user of the TMR HCI to utilize
//during test to interact with the TMR. The GUI will create two additional
//threads - one to set up a serial port connection to listen to the TMR, and
//one to build the data once it is received from the TMR.
//The GUI allow the user to reset the TMR board, or the whole system.
//Author: Susan Groening, LT, USN
//Date: 16 May 00
//*****

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.Color;
import javax.comm.*;
import java.io.*;
import java.util.*;

public class TMRInterfaceClass extends JFrame {

//Class variable to create the GUI components

private final JFrame self;
private JMenuBar menuBar;
private JMenu file;
private JMenuItem exit;
private Container contentPane;
private Box box;
public JButton systemReset;
```

```

public JButton boardReset;
private JLabel welcome;
public JTextArea history;
private JLabel log;
private char mode = 'A';
public String output = " ";
private String filename = " ";

//A Queue object to be utilized to temporarily store the
//register data once the serial port receives it
private FifoBuffer queue = new FifoBuffer();

//byte signals to be used to communicate with TMR
//DUMPFIFO is a future implementation and will not be
//used in this HCI
public static final byte RUNMODEB = 0;
public static final byte RUNMODEA = 1;
public static final byte SYSTEMRESET = 2;
public static final byte BOARDRESET = 4;
//public static final byte DUMPFIFO = 8;

//A thread to set up serial communications with the TMR, listen for data,
//and put the data into the queue
private ReaderThread rt = null;

public TMRInterfaceClass(){

```

```

//create main user window

super("TMR Testbed");
setBackground(Color.gray);
setSize(1025,750);

self = this;

//create a box to hold the components
//the box has vertical layout, components
//added in the order they are created
Box box = Box.createVerticalBox();
box.add(Box.createHorizontalStrut(10));

//create the content pane which all swing components require
contentPane = getContentPane();

//1. create label for the history log
log = new JLabel("TMR History Log");
log.setFont(new Font ("SansSerif", Font.BOLD, 36));
box.add(log);

//2. add a textarea which will display the log
history = new JTextArea(15,15);
box.add(new JScrollPane(history));

//3. add a SYSTEM LEVEL RESET button
systemReset = new JButton("SYSTEM LEVEL RESET");
systemReset.setFont(new Font ("SansSerif", Font.BOLD, 20));

```

```
systemReset.setToolTipText("This will cause a system level reset. System  
level reset and will reset the processors, UART and FIFOs.");
```

```
//this button will reset the processors, UART and FIFOs
```

```
systemReset.addActionListener(new ActionListener(){
```

```
    public void actionPerformed(ActionEvent e)
```

```
    {
```

```
        //dialog box to allow user to resume processing
```

```
        Object[]options={"YES, SYSTEM LEVEL RESET","NO RESET"};
```

```
        int n = JOptionPane.showOptionDialog(self, "Continue with System Level  
Reset?", "RESET CONFIRMATION",  
JOptionPane.DEFAULT_OPTION,JOptionPane.QUESTION_MESSAGE,null,options,o  
ptions[1]);
```

```
        if (n == JOptionPane.YES_OPTION){
```

```
            //send signal to tmr to resume
```

```
            System.out.println("send reset to tmr");
```

```
            System.out.println("Sending Message: " + SYSTEMRESET + "= System  
Level Reset");
```

```
            rt.write(SYSTEMRESET);
```

```
            JOptionPane.showMessageDialog(self,"User initiated system  
reset.", "RESET"
```

```
            ,JOptionPane.PLAIN_MESSAGE);
```

```
            output += "\n User inititated system reset.";
```

```
            history.setText(output);
```

```
            history.setCaretPosition(output.length());
```

```
            //send signal to tmr to stop altogether
```

```
            System.out.println("Resetting system");
```

```
            output += "\n TMR resetting system.";
```

```

        programSaveHistory();
        config();

    }//end if
    else{

        output += "\n No System Level Restart.";
        history.setText(output);
        history.setCaretPosition(output.length());
        System.out.println("No System Level Restart.");

    }//end else
}
}
);
box.add(systemReset);

```

//4. create the board reset button

```

boardReset = new JButton("BOARD LEVEL RESET ");
boardReset.setFont(new Font ("SansSerif", Font.BOLD, 20));
boardReset.setToolTipText("This will reset the FPGA, UART, FIFOs and
processor(s).");

//action for button will reset FPGA, UART, FIFOs and processors. This is the
//most drastic reset available to the user.

```

```

boardReset.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {

```

```

        Object[]options={"YES, BOARD LEVEL RESET","NO BOARD
LEVELRESET."};

```

```
int n = JOptionPane.showOptionDialog(self, "Are you sure you want to reset  
the board? "
```

```
, "RESET  
CONFIRMATION",JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESS  
AGE,null,options,options[1]);
```

```
if (n == JOptionPane.YES_OPTION){  
    System.out.println("Sending Message: " + BOARDRESET +  
"=BOARDRESET");  
    rt.write(BOARDRESET);
```

```
JOptionPane.showMessageDialog(self,"User initiated board  
reset.", "RESET"
```

```
,JOptionPane.PLAIN_MESSAGE);  
output += "\n User initated board reset.";  
history.setText(output);  
history.setCaretPosition(output.length());  
  
System.out.println("Resetting board");  
output += "\n Resetting board.";  
history.setText(output);  
history.setCaretPosition(output.length());  
programSaveHistory();  
config();  
} //end if
```

```
//user selects to continue running the program  
else {
```

```
output += "\n No board reset.";  
history.setText(output);  
history.setCaretPosition(output.length());
```

```
System.out.println("Continuing program");  
output += "\n Continuing program.";
```

```

        history.setText(output);
        history.setCaretPosition(output.length());
    }

}

};

box.add(boardReset);

//add box to content pane
contentPane.add(box);

//create the File menu for the GUI
//create a menu bar
menuBar = new JMenuBar();

//create a file menu
file = new JMenu("File");
file.setMnemonic('F');

//create an 'exit' option
exit = new JMenuItem("Exit");
exit.setMnemonic('X');

//add exit to File menu
file.add(exit);
exit.addActionListener (new exitHandler());

//add File to menu bar
menuBar.add(file);

```



```

//set the menubar for the frame
setJMenuBar(menuBar);

addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e){
        output += "\n Exiting";
        history.setText(output);
        history.setCaretPosition(output.length());
        System.err.println("Exiting Gracefully.");
        System.exit(0);
    }
    public void windowClosed(WindowEvent e){
        e.getWindow().setVisible(false);
    }
});

//make reader Thread
rt = new ReaderThread("COM1",queue);
rt.start();

//make a buildData thread
buildDataThread bdt = new buildDataThread(this,queue);
bdt.start();

} //end guiClass constructor

```

```

//*****
//Name: config()
//Return : void
//Parameter: none
//Purpose: Brings the user interface window to the screen to allow the
//user to select which processor mode they want to use.
//*****

```

```

public void config() {
    do {
        selectProcessorMode();
    } while (!verifyRun());

} //end config()

```

```

//*****
//Name: setVisible
//Return: void
//Parameter: boolean
//Purpose: Sets the GUI visible once the program has been instantiated. This
//calls the opening welcome dialog box, then calls config to allow processor
//mode selection
//*****

```

```

public void setVisible(boolean show) {
    super.setVisible(show);
    if (show) {
        welcome();
        config();
    }
}

```

```

        }//end if
    }//end setVisible

    /*******
    //Name: welcome()
    //Returns: void
    //Parameter: none
    //This function displays a dialog box that states the user is going to use the TMR
    //testbed. User clicks OK and continues on. If user selects cancel, the program
    //exits.
    /*******

    public void welcome() {

        int n = JOptionPane.showConfirmDialog(self, "Welcome to the TMR
Testbed."+
        " Press OK to continue, or Cancel to exit."

        "Welcome",JOptionPane.OK_CANCEL_OPTION,JOptionPane.PLAIN_MESSAGE);

        //if statement to perform an action depending on user's selections
        if (n == JOptionPane.CANCEL_OPTION){
            output += "\n Exiting";
            history.setText(output);
            history.setCaretPosition(output.length());
            System.exit(0);
        }//end if

    }//end welcome()

```

```

/*****/
//Name: selectProcessorMode()
//Returns: void
//Parameter: none
//This function allows the user to select Mode A, which will run all three
//processors at the same time, or Mode B, which will run only the radiated
//processor.
/*****/

public void selectProcessorMode(){
    //dialog window which offers up two choices, A or B
    Object[] options={"Mode A - Run 3 processors","Mode B - Run single processor
    "};

    int n = JOptionPane.showOptionDialog(self, "Select Processor mode.\n"+
        " Only one processor will be radiated.\n"+
        "(After selecting mode, use VxWorks interface to load and run program.)",
        "PROCESSOR
MODE",JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,null,options,options[1]);

    //if statement to take action depending on user's selection
    if (n == JOptionPane.YES_OPTION){
        System.out.println("Sending Message: " + RUNMODEA + "= Mode A
selected");
        rt.write(RUNMODEA);
        mode = 'A';

        System.out.println("Mode A selected");
        output += "\n Mode A selected.";

        history.setText(output);
    }
}

```

```

        history.setCaretPosition(output.length());
    } //end if

    else {
        System.out.println("Sending Message: " + RUNMODEB + "= Mode B
selected");
        rt.write(RUNMODEB);

        mode = 'B';
        System.out.println("Mode B selected");
        output += "\n Mode B selected.";
        history.setText(output);
        history.setCaretPosition(output.length());

    } //end else

} //end selectProcessorMode()

```

```

/*****
//Name: verifyRun()
//Returns: boolean
//Parameter: none
//This function will verify that the user has selected the run option. If they do
//not select run, it will return them to the select processor mode
*****/

```

```

public boolean verifyRun(){

```

```

    Object[] options={"Continue","Back"};

```

```

int n = JOptionPane.showOptionDialog(self, "Mode " + mode + " selected.\n" +
    " Ensure your program is loaded using VxWorks.\n"+
    " Press Continue to proceed with testing, or Back to change mode"
    ,"Verify Processor Mode and Program
Loaded",JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,null,options,options[1]);

if (n == JOptionPane.YES_OPTION){
    output += "\n Listening for register data.";
    JOptionPane.showMessageDialog(self,"Execute your application now "+
        " from the VxWorks interface.\n","EXECUTE PROGRAM"
        ,JOptionPane.PLAIN_MESSAGE);

    history.setText(output);
    history.setCaretPosition(output.length());
    return true;
} //end if

else {
    return false;
} //end else

} //end verifyRun

```

```

//*****
//Name: programSaveHistory()
//Returns: void
//Parameters: none
//This program allows the user to save the History text file from the run just
//completed.
//*****

```

```

private void programSaveHistory(){
    File newFile;
    FileOutputStream textfos;
    OutputStreamWriter textosw;
    BufferedWriter textbw;
    PrintStream textps;

    String historyLog = "";
    InputStreamReader fileisr;
    FileInputStream someLogOutput;
    BufferedReader someTextbr;

    String logString = "";

    Object [] optionsC = {"YES, SAVE HISTORY LOG","NO, DISREGARD
HISTORYLOG"};

    int n = JOptionPane.showOptionDialog(self,"Do you want to save the history
log?\n (Aseparate prompt will give you the option to save register data.)", "SAVE
HISTORY IFORMATION",

JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,null,optionsC,pt
ionsC[1])

    if (n == JOptionPane.YES_OPTION){
        JFileChooser saveFileChooser = new JFileChooser("E:\\");
        try{
            int returnValue = saveFileChooser.showSaveDialog(this);
            if(returnValue == JFileChooser.APPROVE_OPTION){
                File fileName = saveFileChooser.getSelectedFile();
                PrintStream ps = new PrintStream(new FileOutputStream(fileName));
                boolean afile = fileName.exists();
                System.out.println(afile + "does it exist");//with this cofig, yes

```

```

        boolean awrite = fileName.canWrite();
        System.out.println(awrite + " can write");
        String aname = fileName.getPath();
        System.out.println(aname + " is path/filename");

        StringReader stringReader = new StringReader(history.getText());
        someTextbr = new BufferedReader(stringReader);

        logString = someTextbr.readLine();
        while(logString != null){

            ps.println(logString);
            logString = someTextbr.readLine();
        }
        someTextbr.close();

    } //end if
}

catch(IOException e){
    System.err.println( e.toString());
}

output += "\n" + historyLog + "saved.";
history.setText(output);
history.setCaretPosition(output.length());
} //end if
else{
    String historyLogA = "";
    Object[] optionsD = {"GO BACK TO SAVE HISTORY LOG", "DISCARD HISTORY LOG"}
    int xx = JOptionPane.showOptionDialog(self, "WARNING: Clicking 'Discard HistoryLog' will cause history log to be lost.", "DISREGARD HISTORY LOG",

```



```
JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,null,optionsD,ptionsD[1])
```

```
    if (xx == JOptionPane.NO_OPTION){
```

```
        }//do nothing end if
```

```
    else{
```

```
        JFileChooser saveFileChooser = new JFileChooser("E:");
```

```
        try{
```

```
            int returnValue = saveFileChooser.showSaveDialog(this);
```

```
            if(returnValue == JFileChooser.APPROVE_OPTION){
```

```
                File fileName = saveFileChooser.getSelectedFile();
```

```
                PrintStream ps = new PrintStream(new FileOutputStream(fileName));
```

```
                boolean afile = fileName.exists();
```

```
                System.out.println(afile + "does it exist");//with this cofig, yes
```

```
                boolean awrite = fileName.canWrite();
```

```
                System.out.println(awrite + " can write");
```

```
                String aname = fileName.getPath();
```

```
                System.out.println(aname + " is path/filename");
```

```
                StringReader stringReader = new StringReader(history.getText());
```

```
                someTextbr = new BufferedReader(stringReader);
```

```
                logString = someTextbr.readLine();
```

```
                while(logString != null){
```

```
                    ps.println(logString);
```

```
                    logString = someTextbr.readLine();
```

```
                }
```

```
                someTextbr.close();
```

```
            }//end if
```

```
        }//end try
```

```
    catch(IOException e){
```

```

        System.err.println( e.toString());
    }
    }//end else
} //end else
return;

} //end programSaveHistory

//*****
//Name: main
//Return: void
//Parameter: String{ } args
//This is main. It calls the GUI constructor and then call setVisible()
//*****

public static void main(String[] args){

    TMRInterfaceClass HCI = new TMRInterfaceClass();
    HCI.setVisible(true);

} //end main

} //end guiClass Class

```

THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX B. READERTHREAD.JAVA

```
//ReadThread.java
//A class that instantiates a thread to set up serial communications
//during testing to interact with the TMR. The thread listens for data
//on the serial port, stores the received data in a queue, and then
//wakes up another thread to calculate the bit flip.
//Author: Susan Groening, LT, USN
//Date: 16 May 00
//*****

import java.io.*;
import java.util.*;
import javax.comm.*;

public class ReaderThread extends Thread implements SerialPortEventListener {

    //class Variables
    private boolean running = true; //keeps the thread alive
    private FifoBuffer queue; //a queue object
    public InputStream inputStream;
    public OutputStream outputStream;
    public SerialPort serialPort;
    public byte b;

    public ReaderThread(String port, FifoBuffer buf) {

        queue = buf;
        CommPortIdentifier portId = null;
```

```

//identify the port to be used for the communications
try {
    portId = CommPortIdentifier.getPortIdentifier(port);
} catch (NoSuchPortException nex) {
    System.out.println(nex);
}

//try to open up the port
try {
    System.out.println("Attempting to open port: " + port);
    serialPort = (SerialPort) portId.open("ReaderThread", 2000);
    System.out.println("Opened port: " + port);
} catch (PortInUseException e) {
    System.out.println(e);
}

//establish input and output streams
try {
    inputStream = serialPort.getInputStream();
    outputStream = serialPort.getOutputStream();
} catch (IOException e) {
    System.out.println(e);
}

//add an event listener to listen for data on the port
try {
    serialPort.addEventListener(this);
} catch (TooManyListenersException e) {
    System.out.println(e);
}

serialPort.notifyOnDataAvailable(true);
//set the port parameters
try {

```

```

        serialPort.setSerialPortParams(9600,
            SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1,
            SerialPort.PARITY_NONE);
    } catch (UnsupportedCommOperationException e) {
        System.out.println(e);
    }
} //end ReaderThread

```

```

//*****
//Name: quit
//Return: void
//Parameter: none
//Purpose: A method to facilitate a way to stop the thread from running
//as needed. It is not implemented in ReaderThread
//*****

```

```

// public void quit() {
//     running = false;
// } //end ReaderThread

```

```

//*****
//Name: run
//Return: void
//Parameter: none
//Purpose: A method to start a thread running.
//*****

```

```

public void run() {
    while (running) {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
        }
    } //end while
} //end run

```

```

//*****
//Name: write
//Return: void
//Parameter: byte
//Purpose: A method to write a byte from the HCI to the TMR.
//*****

```

```

public void write(byte b) {
    try {
        outputStream.write(b);

    } catch (IOException ex) {
    }
} //end write

```

```

//*****
//Name: serialEvent
//Return: void
//Parameter: SerialPortEvent
//Purpose: A method to handle serial port events.
//*****

```

```

public void serialEvent(SerialPortEvent event) {
    //any event that the serial port listener detects is passed to this
    //method which then runs the event through the switch to determine
    //how to handle the particular event
    switch(event.getEventType()) {
        case SerialPortEvent.BI:
        case SerialPortEvent.OE:
        case SerialPortEvent.FE:
        case SerialPortEvent.PE:
        case SerialPortEvent.CD:
        case SerialPortEvent.CTS:
        case SerialPortEvent.DSR:
        case SerialPortEvent.RI:
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            break;
        case SerialPortEvent.DATA_AVAILABLE:
            try {
                if (inputStream.available() > 0) {
                    //synchronized is used to provide mutual exclusion
                    //to the queue during operations on the queue
                    synchronized(queue) {

                        while (inputStream.available() > 0) {
                            b = (byte) inputStream.read();

```



```

        queue.put(b);
    } //end while
    queue.notifyAll(); //wakes up any threads waiting on the queue
} //end lock

} //end if
} catch (IOException e) {
    System.out.println(e);
}
break;

} // end switch
} //end serialEvent()
} //end Class

```

## APPENDIX C. BUILD DATATHREAD

```
//buildDataThread.java
//A class that instantiates a thread to find the error in the TMR data.
//This thread will take the received data out of a queue, put it into
//arrays, then locate the flip in the array. It will maintain the history
//log, as well as store the data into an Access database.
//Author: Susan Groening, LT, USN
//Date: 16 May 00
//*****

import java.util.*;
import java.util.Date;
import java.io.*;
import java.sql.*;

public class buildDataThread extends Thread {

    //class variables
    private FifoBuffer queue;
    private TMRInterfaceClass HCI;

    //variables to create the three arrays that will store the error data
    //The size of the arrays can be changed, if the number of entries per
    //each register increases or decreases.
    public static final int NUM_REG = 41;
    int registerData[][] = new int[3][NUM_REG];
    int addresses[][] = new int[3][NUM_REG];
    byte control[][] = new byte[3][NUM_REG*2];
```

```

public boolean running = true;

//Connection con is the variable to establish a connection from the HCI to
//the Access database
private Connection con;

//Statement stmt is the variable used to make a Create Statement to update
//the Access database tables
private Statement stmt;

//variables used to transmit information to the database in SQL queries
public String url;
private int time= 0;
private char errorProcessor;
private int errorRegister;
private int errorBit;
public int errorCount=0;
int bigCount = 0;
String key = " ";
String logTime = "";

public buildDataThread(TMRInterfaceClass gui, FifoBuffer buf) {
    queue = buf;
    HCI = gui;

} //end buildData()

public synchronized void buildData() { // throws InterruptedException {

```

```
key = (new Date()).toString();//get the current date and time to create
//a key for the database
```

```
logTime = key;//logtime will be used to display the time/date to the
//History Log
```

```
//using nested for loops, iterate through them and get a byte from
//the queue for each array element
```

```
//test://label to utilize a named break statement to exit if reset occurs
// to be utilized during future reset implementation
for (int i = 0; i < 3; i++) { //outer for loop will count the processors
```

```
for (int j = 0; j < 5; j++) { //middle for loop counts the fifos
```

```
byte header = queue.get(); //get the header byte from the queue
for (int k = 0; k < NUM_REG * 2; k++) { //the inner for loop will
//be used to populate the array
```

```
int regIndex = k / 2;
```

```
/*if (HCI.killIt){ //break out of loop if reset has occurred
System.out.println("Exiting for loop after reset");
clearArrays();
break test;
}
```

```
this for loop will be utilized once the reset capability has
been implemented.
```

```
*/
```

```

int b = queue.get();

b = b & 0xFF; //gets rid of the negative sign
if (j == 4) {
    control[i][k] = (byte) b; //this would be the control byte and
    //we are not interested in that data
    //at this stage of the HCI development
} //end if
else if ((k & 1) == 1) { //in the bit stream, the data items are
    //in the odd numbered places (1,3,5,7)
    //Therefore, if the data & 1 == 1, then
    //the bit stream location is odd
    //(& equals 1 when both are the same)
    //This checks to see if we are storing
    //addresses or data
    if (j == 0) {
        registerData[i][regIndex] = b; //puts the first byte of the
        //number into the array
    }
    else {
        registerData[i][regIndex] = (registerData[i][regIndex] << 8) | b;
    }
    //if the number being put into the array
    //is the 2nd, 3rd or 4th byte, then the
    //existing number in the array is
    //shifted left 8 bits and inclusive or (!)
    //with the number already in the array.
    //this builds the number into an integer
} //end else if
else {
    if (j == 0) {
        //The number came from an even bit location
        //in the register, and is therefore an

```

```

                                //address location
                addresses[i][regIndex] = b;
            }
            else {
                addresses[i][regIndex] = (addresses[i][regIndex] << 8) | b;
            }
        } //end else
    } //end inner for

} //end middle for
} //end outer for

} //end buildData

/*****
//Name: run()
//Return: void
//Parameter : none
//Purpose: To keep the thread running during program execution. Build data
//retrieves the data from the queue and stores it into an array. findFlip
//locates the bit flip, and storeData transfers data to the database
*****/

public void run() {

    while (running){

        buildData();
        findFlip();
        storeData();
    }
}

```

```

        //displayDataTest(); can display the register contents to the screen
        //if desired
    }//end while

} //end run()


//*****
//Name: quit()
//Return: void
//Parameter: none
//Purpose: to cause the thread to die
//*****
public void quit(){

    running = false;
} //end quit


//*****
//Name: findFlip()
//Return: void
//Parameter: none
//Purpose: to locate the single bit flip in the register data
//*****
public void findFlip(){

```

```
//toggle to set the error count to 1 whenever user resets the processors or the  
//board and executes another test run.
```

```
if (HCI.toggle == false){  
    errorCount = 0;
```

```
    HCI.toggle = true;  
} //end if
```

```
int i,j,k;
```

```
for (i = 0; i < 1 ; i++){  
    for (k = 0; k < NUM_REG; k++){  
        int regIndex = k;  
        if ((registerData[i][regIndex]^registerData[i+1][regIndex]) == 0){  
            if((registerData[i+1][regIndex]^registerData[i+2][regIndex]) == 0){  
                //HCI.output += "No error was found; ";  
                //HCI.history.setText(HCI.output);  
                //HCI.history.setCaretPosition(HCI.output.length());  
            } //then no error
```

```
        else{ //error in Processor C
```

```
            int error = registerData[i+1][regIndex]^registerData[i+2][regIndex];  
            int bitCount = 0;
```

```
            //the exclusive or operator will return int error, which will have  
            //a one, where there is a difference between the two bits in the arrays.  
            //Iterating through error and performing an & operation on error  
            //and 1, locates the bit that was flipped. The & operator will return a 1  
            //only if the bit that was flipped has been found.
```



```

while ((error & 1)==0){
    bitCount++;
    error = error >> 1;//right bitshift to examine the next bit
} //end while
errorCount ++;
HCL.output += "\n" + logTime + " Error # " + errorCount + " : Error
found processor C, register " + k + ", bit " + bitCount + ".";

```

```

HCL.history.setText(HCL.output);
HCL.history.setCaretPosition(HCL.output.length());

```

```

errorProcessor = 'C';
errorRegister = k;
errorBit = bitCount;

```

```

//HCL.output += " Error is in bit : " + bitCount;

```

```

// HCL.history.setText(HCL.output);
// HCL.history.setCaretPosition(HCL.output.length());

```

```

//create a connection with the Access database

```

```

try{
    url = "jdbc:odbc:Registers";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

```

```

    con = DriverManager.getConnection (url,"anonymous","guest");
    System.out.println("find flip db connection successful\n");
    //this for loop grabs the contents of each array element, which
    //is a register data item, so it can be included as register data
    //in the database

```

```

        for (int ed = 0; ed < NUM_REG; ed++){

            int dataNumber = registerData[i+2][ed];
            Statement stmt = con.createStatement();
            int result = stmt.executeUpdate("INSERT INTO Registers VALUES
(""+key+"",""+bigCount+"",""+dataNumber+"");
            bigCount++;
        }
        con.close();
    }
    catch(ClassNotFoundException cnfex){
        cnfex.printStackTrace();
    }
    catch (SQLException sqlex){
        sqlex.printStackTrace();
    }
    catch (Exception ex){
        ex.printStackTrace();
    }

    } //end else
} //end if

else { //error was in processor B or A
    if ((registerData[i][regIndex]^registerData[i+2][regIndex]) == 0){ //Error
is in B

        int errorValue =
(registerData[i][regIndex]^registerData[i+1][regIndex]);
        int bitCount = 0;
        while ((errorValue & 1) == 0) {
            bitCount++;

```

```

        errorValue = errorValue >> 1;
    }//end while

    errorCount++;

    HCI.output += "\n " + logTime + "Error # " + errorCount + " : Error
found processor B, register " + k + ", bit " + bitCount + ".";

    HCI.history.setText(HCI.output);
    HCI.history.setCaretPosition(HCI.output.length());
    errorProcessor = 'B';
    errorRegister = k;
    errorBit = bitCount;
    //send data to Access Database
    try{
        url = "jdbc:odbc:Registers";
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        con = DriverManager.getConnection (url,"anonymous","guest");
        System.out.println("find flip db connection successful\n");

        for (int ed = 0; ed < NUM_REG;ed++){
            bigCount++;
            int dataNumber = registerData[i+1][ed];
            Statement stmt = con.createStatement();
            int result = stmt.executeUpdate("INSERT INTO Registers VALUES
("+key+", "+bigCount+", "+dataNumber+)");
        }
        con.close();
    }
    catch(ClassNotFoundException cnfex){
        cnfex.printStackTrace();
    }

```

```

        catch (SQLException sqlex){
            sqlex.printStackTrace();
        }
        catch (Exception ex){
            ex.printStackTrace();
        }

    }//end if

    else{ //error was in processor A
        errorCount++;

        int errorValue = (registerData[i][regIndex]^registerData[i+1][regIndex]);
        int bitCount = 0;
        while ((errorValue & 1) == 0) {
            bitCount++;
            errorValue = errorValue >> 1;
        }//end while

        HCI.output += "\n" + logTime + " Error # " + errorCount + " : Error
found processor A, register " + k + ", bit " + bitCount + ".";
        HCI.history.setText(HCI.output);
        HCI.history.setCaretPosition(HCI.output.length());
        errorProcessor = 'A';
        errorRegister = k;
        errorBit = bitCount;
    }//end else

    //HCI.output += " Error is in bit : " + bitCount;
    //HCI.history.setText(HCI.output);
    //HCI.history.setCaretPosition(HCI.output.length());

```

```

//send register data from error Processor to Access data base
try{
    url = "jdbc:odbc:Registers";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    con = DriverManager.getConnection (url,"anonymous","guest");
    System.out.println("find flip db connection successful\n");

    for (int ed = 0; ed < NUM_REG;ed++){
        bigCount++;
        int dataNumber = registerData[i][ed];
        Statement stmt = con.createStatement();
        int result = stmt.executeUpdate("INSERT INTO Registers VALUES
("+key+"','"+bigCount+"','"+dataNumber+"")");
    }
    con.close();
}
catch(ClassNotFoundException cnfex){
    cnfex.printStackTrace();
}
catch (SQLException sqllex){
    sqllex.printStackTrace();
}
catch (Exception ex){
    ex.printStackTrace();
}

} //end A else
} //end else
} //end for
} //end for

```

```

} //end findFlip

//*****
//displayDataTest()
//Return: void
//Parameter: none
//This function will allow the contents of the register to be displayed to the
//command line screen if the user desires. It is not used in the current version
//of the HCI. However, it could be called following a call to findFlip
//*****

public void displayDataTest()
{

    for (int i = 0; i < 3; i ++){ //processors
        if(i == 0){
            System.out.println("processor A DATA");
            HCI.output += "\n Processor A Data";
            HCI.history.setText(HCI.output);
            HCI.history.setCaretPosition(HCI.output.length());
        }

        else if (i == 1){
            HCI.output += "\n Processor B Data";
            HCI.history.setText(HCI.output);
            HCI.history.setCaretPosition(HCI.output.length());
            System.out.println("processor B DATA");
        }

        else{

```

```

        HCI.output += "\n Processor C Data";
        HCI.history.setText(HCI.output);
        HCI.history.setCaretPosition(HCI.output.length());
        System.out.println("Processor C Data");
    }

    for (int k = 0; k < NUM_REG; k++){
        int regIndex = k;
        System.out.print( " " + (registerData[i][regIndex]));
        HCI.output += (registerData[i][regIndex]) + " ";
        HCI.history.setText(HCI.output);
        if (regIndex%6 == 0){

            HCI.output += "\n";
            HCI.history.setText(HCI.output);
            HCI.history.setCaretPosition(HCI.output.length());
            System.out.print("\n");
        } //end if

    } //end for

} //end dataDisplayTest()

```

```

/*****

```

```

//Name: storeData

```

```

//Parameter: none

```

```

//Return: void

```

```

//This method will open a connection to the Access Database, and store the

```

```

//time, processor, register and bit that experienced an error into the TMRDATA
//database.
//*****

public void storeData()
{

    try{
        url = "jdbc:odbc:TMRData";
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        con = DriverManager.getConnection (url,"anonymous","guest");
        System.out.println("store data db connection successful\n");
        Statement stmt = con.createStatement();

        int result = stmt.executeUpdate("INSERT INTO TMRData VALUES
("+key+", "+errorProcessor+", "+errorRegister+", "+errorBit+"");//TIME,errorProcessor,
errorRegister,errorBit");

        con.close();
    }
    catch(ClassNotFoundException cnfex){
        cnfex.printStackTrace();
    }
    catch (SQLException sqlex){
        sqlex.printStackTrace();
    }
    catch (Exception ex){
        ex.printStackTrace();
    }

}

} //end storeData()

```



```

/*****
//Name: clearArrays
//Parameter: none
//Return: void
//This method is to be utilized in future reset implementation.
//It will set all array values to zero. It is used after a reset
//to permit the find flip algorithm to start with a clean array
*****/

    public synchronized void clearArrays()
    {

        for (int i = 0; i < 3; i++){
            for (int k = 0; k < NUM_REG; k++){
                int regIndex = k;
                registerData[i][regIndex] = 0;
                addresses[i][regIndex] = 0;
            }
        }
    } //end clearArrays

} //end class

```

## APPENDIX D. FIFOBUFFER.JAVA

```
//FifoBuffer.java
//A class that instantiates a fifo queue for the HCI to utilize to temporarily
//store error data when it is received over the serial port. The class uses
//synchronized methods to provide a mutual exclusion lock during get and
//put operations.
//Author: Susan Groening, LT, USN
//Date: 16 May 00
//*****

import java.util.*;

public class FifoBuffer {

    private Vector queue = new Vector();

    //*****
    //Name: put
    //Parameter: byte
    //Return: void
    //Purpose is to put a byte of data into the queue
    //*****

    public synchronized void put(byte data) {

        queue.add(new Byte(data));

    } //end put
```

```

//*****
//Name: get
//Parameter: None
//Return: byte
//Purpose: to get one byte of information from the queue
//*****

```

```

public synchronized byte get() {

    while (queue.size() == 0) {
        try {
            wait();
        } catch (InterruptedException ex) {
        }
    }
    }//end while

    Byte b = (Byte) queue.remove(0);
    return b.byteValue();

} //end get

```

```

//*****
//Name: isEmpty
//Parameter: None
//Return: boolean
//Purpose: returns true if the queue is empty.
//*****

```

```

public synchronized boolean isEmpty(){

```

```

        boolean empty = false;
        empty = queue.isEmpty();
        return empty;
    }//end isEmpty

```

```

//*****
//Name: getLength
//Parameter: None
//Return: int
//Purpose: returns the queue length.
//*****

```

```

public int getLength(){
    int size = queue.size();
    return size;
} //end getLength

```

```

//*****
//Name: emptyQueue
//Parameter: None
//Return: void
//Purpose: empties the queue
//*****
public synchronized void emptyQueue(){
    queue.removeAllElements();
}

```

```
return;
```

```
//end emptyQueue
```

```
//end FifoBuffer Class
```

## APPENDIX E. TMRTESTPANEL.JAVA

//TMRTestPanel Class

//Purpose: At user's discretion, simulates register data that is transmitted to the  
//HCI following a TMR- recognized error. User prompts program to send error  
// data.

//Author: LCDR C. Eagle, USN

//Modified by LT S. Groening, USN, to transmit continuous errors to the HCI.

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import java.io.\*;

import javax.comm.\*;

import java.util.\*;

public class TmrTestPanel extends JFrame implements SerialPortEventListener {

public static final byte MODE\_B = 0;

public static final byte MODE\_A = 1;

public static final byte BOARD\_RESET = 4;

public static final byte SYSTEM\_RESET = 2;

public static String port = "";

public static int randProc;

public static int randTime;

public static boolean testing = true;

public int tester = 0;

private static final byte NUM\_REG = 41;

private char procs[] = {'A', 'B', 'C'};

private Random rand;

```

private boolean running = true;
private JPanel buttonPanel = new JPanel();
private int registers[] = new int[NUM_REG];
private OutputStream outputStream;
private InputStream inputStream;
private SerialPort serialPort;

```

//constructor to create a small GUI the user can signal errors to the HCI

```

public TmrTestPanel(String port) {
    super("Tmr Control Panel");
    initPort(port);
    rand = new Random(hashCode());

    buttonPanel.setLayout(new GridLayout(1, 4));

    JButton button;

    button = new JButton("QUIT");
    button.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                running = false;
                System.exit(0);
            }
        }
    );
    buttonPanel.add(button);

    button = new JButton("Error A");
    button.addActionListener(

```

```
new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        generateError(0);  
    }  
}  
);  
buttonPanel.add(button);
```

```
button = new JButton("Error B");  
button.addActionListener(  
    new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            generateError(1);  
        }  
    }  
);  
buttonPanel.add(button);
```

```
button = new JButton("Error C");  
button.addActionListener(  
    new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            generateError(2);  
        }  
    }  
);  
buttonPanel.add(button);
```

```
setContentPane(buttonPanel);
```

```
setSize(400, 75);
```



```

        setVisible(true);
    }//constructor

```

```

//*****
//Name: generateError
//Return: void
//Parameter: int
//Purpose: Method used to generate the error
//*****

```

```

private void generateError(int processor) {
    tester++;//variable to display the error number to the screen

    //randomly select a processor, register and bit
    byte badReg = (byte) rand.nextInt(NUM_REG);
    byte fifoNum = (byte) rand.nextInt(4);
    byte bit = (byte) rand.nextInt(8);
    byte bitNum = (byte) (fifoNum * 8 + bit);
    int startAddress = rand.nextInt(4096 - NUM_REG);
    //fill registers with data:
    for (int i = 0; i < NUM_REG; i++) {
        registers[i] = rand.nextInt();
    }
    System.out.println("This is error number: " + tester);
    System.out.println("Sending data. Error on processor: " + procs[processor]);
    System.out.println("Error is in register: " + badReg + ", bit: " + bitNum);
    System.out.println("Start address is: " + startAddress);
    System.out.println();
    try {
        //loop for each of the three processors
        for (byte proc = 0; proc < 3; proc++) {

```

```

//initial shift is 24 bits for fifo zero
byte shift = 24;

//count down through the four fifos for each register
for (byte fifo = 3; fifo >= 0; fifo--) {

    //generate the header that precedes each fifo's data
    byte header = (byte) (proc + (3 - fifo) * 4);
    outputStream.write(header);

    int address = startAddress;

    for (byte reg = 0; reg < NUM_REG; reg++) {
        //first output the appropriate byte of the address
        byte add = (byte) (address >> shift);
        outputStream.write(add);

        //then output the appropriate byte of the register
        byte data = (byte) (registers[reg] >> shift);
        if (reg == badReg && fifo == fifoNum && proc == processor) {
            //if this is the register that is supposed to be bad
            //and we are writing the appropriate byte in that register
            //then flip the selected bit to generate an error
            data = (byte) (data ^ (1 << bit));
        }
        outputStream.write(data);

        address++;
    }
}

```

```

        }
        //reduce shift amount for successive fifos
        shift -= 8;
    }

    //now send the control fifo
    byte header = (byte) (proc + 16);
    outputStream.write(header);

    for (byte reg = 0; reg < NUM_REG * 2; reg++) {
        outputStream.write(0);
    }

}

} catch (Exception ex) {

}

} //end generateError

```

```

//*****
//Name: initPort
//Return: void
//Parameter: String
//Purpose: establish a serial port
//*****

```

```

public void initPort(String port) {
    CommPortIdentifier portId = null;

```

```

try {
    portId = CommPortIdentifier.getPortIdentifier(port);
} catch (NoSuchPortException nex) {
    System.out.println(nex);
}
try {
    System.out.println("Attempting to open port: " + port);
    serialPort = (SerialPort) portId.open("ReaderThread", 2000);
    System.out.println("Opened port: " + port);
} catch (PortInUseException e) {
    System.out.println(e);
}
try {
    inputStream = serialPort.getInputStream();
    outputStream = serialPort.getOutputStream();
} catch (IOException e) {
    System.out.println(e);
}
try {
    serialPort.addEventListener(this);
} catch (TooManyListenersException e) {
    System.out.println(e);
}
serialPort.notifyOnDataAvailable(true);
try {
    serialPort.setSerialPortParams(9600,
        SerialPort.DATABITS_8,
        SerialPort.STOPBITS_1,
        SerialPort.PARITY_NONE);
} catch (UnsupportedCommOperationException e) {
    System.out.println(e);
}

```

```

    }
} //initPort

```

```

//*****
//Name: serialEvent
//Return: void
//Parameter: SerialPortEvent
//Purpose: handle the serial port events
//*****

```

```

public void serialEvent(SerialPortEvent event) {
    switch(event.getEventType()) {
        case SerialPortEvent.BI:
        case SerialPortEvent.OE:
        case SerialPortEvent.FE:
        case SerialPortEvent.PE:
        case SerialPortEvent.CD:
        case SerialPortEvent.CTS:
        case SerialPortEvent.DSR:
        case SerialPortEvent.RI:
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            break;
        case SerialPortEvent.DATA_AVAILABLE:
            try {
                while (inputStream.available() > 0) {
                    byte b = (byte) (inputStream.read() & 7);
                    if ((b & BOARD_RESET) == BOARD_RESET) {
                        System.out.println("Received \"board reset\" command");
                        running = false; //to stop data generation
                        System.exit(0); // = false; //to stop testing
                    }
                }
            }

```

```

        else if ((b & SYSTEM_RESET) == SYSTEM_RESET) {
            System.out.println("Received \"system reset\" command");
            running = false; //to stop data generation
            System.exit(0); //testing = false; //to stop testing
        }
        else if (b == MODE_B) {
            System.out.println("Received \"run mode B\" command");
        }
        else {
            //process of elimination
            System.out.println("Received \"run mode A\" command");
        }
    }
} catch (IOException e) {
    System.out.println(e);
}
break;
}
} //end SerialPortEvent

```

```

//*****

```

```

//Name: Main

```

```

//Return: void

```

```

//Parameter: String

```

```

//Purpose: instantiate the GUI and, if desired, the ability to create

```

```

//continuous errors, or a fixed number of randomly generated errors

```

```

//*****

```

```

    public static void main(String args[]) {

```

```

//implement with this if using command line arguments.
/*if (args.length != 1) {
    System.out.println("Usage: java TmrTestPanel portName");
    System.exit(0);
}
*/

TmrTestPanel.port = "COM1";
TmrTestPanel tmr = new TmrTestPanel(port);

/*// while(testing){
//system test
1st For loop will generate 10 errors, spaced randomly apart
for (int ix = 0; ix <2; ix++){
    randProc = (int)(Math.random()*3);//0 = Proc A, 1 = Proc B, 2 = Proc C
    randTime = (int)(Math.random()*10000000);

    for(int cix = 0; cix <10000000;cix++){
        if(randTime == cix){ //will generate error at randTime
            tmr.generateError(randProc);
        }//end if
    }//end for

}

}

//generate rapidly occuring errors
for (int ix = 0; ix <20; ix++){
    randProc = (int)(Math.random()*3);//0 = Proc A, 1 = Proc B, 2 = Proc C
    randTime = (int)(Math.random()*100);
}
}

```

```

for(int cix = 0; cix <100;cix++){
    if(randTime == cix){ //will generate error at randTime
        tmr.generateError(randProc);
    }//end if
} //end for

//generate rapid errors
for(int cix = 0; cix <20;cix++){
    randProc = (int)(Math.random()*3);//0 = Proc A, 1 = Proc B, 2 = Proc C
    tmr.generateError(randProc);
} //end for

//generate random errors spaced apart
for (int ix = 0; ix <5; ix++){
    randProc = (int)(Math.random()*3);//0 = Proc A, 1 = Proc B, 2 = Proc C
    randTime = (int)(Math.random()*1000000000);

    for(int cix = 0; cix <1000000000;cix++){
        if(randTime == cix){ //will generate error at randTime
            tmr.generateError(randProc);
        } //end if
    } //end for
} //end for

//} //end while
*/

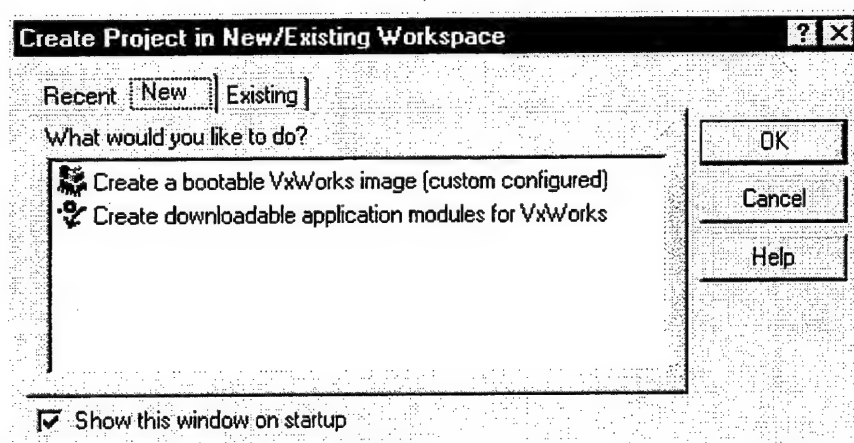
```



```
}//end Main  
}//end Class
```

## APPENDIX F. BUILDING A PROJECT FROM A NEW BSP

1. Place all required files in the `target/config/bspname` Directory. Ensure all required routines are included (for a list of required files and routines, see Tornado BSP Developer's Kit for VxWorks).
2. Select `Start ⇒ Programs ⇒ Tornado2 ⇒ Tornado`
3. `File ⇒ New Project` will bring up the following window:



Select "Create a bootable VxWorks image (custom configured)" and hit OK.

4. The following window will appear:

**Create a bootable VxWorks image [custom configured]: step 1**

**Project**

Name:

Location:

**Project description (optional)**

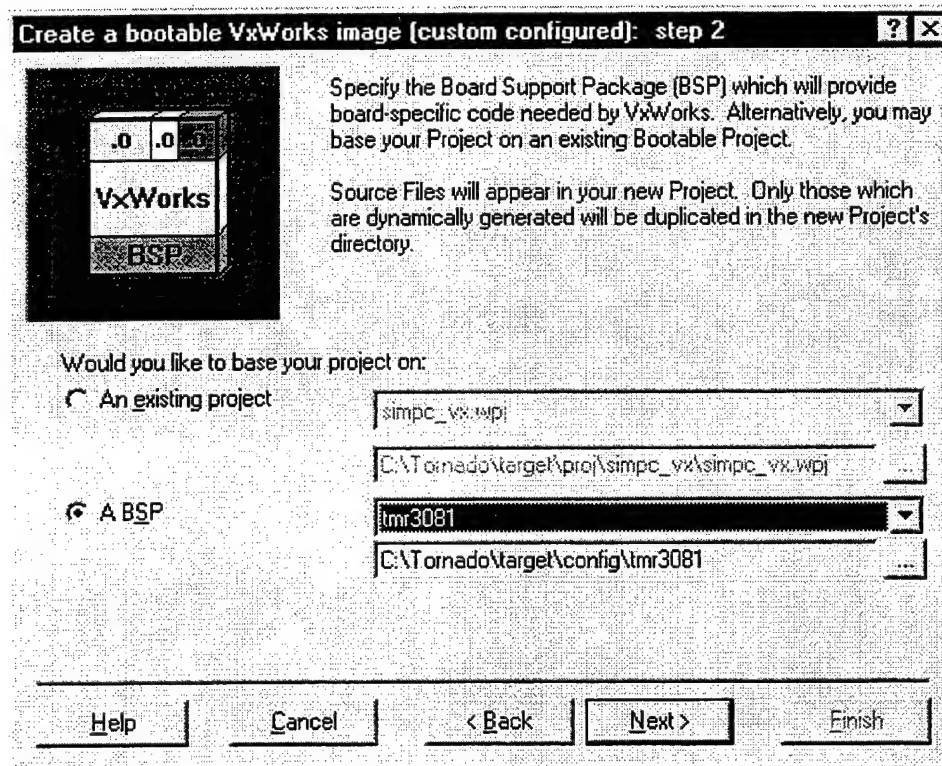
**Workspace**

☐ Add to current Workspace

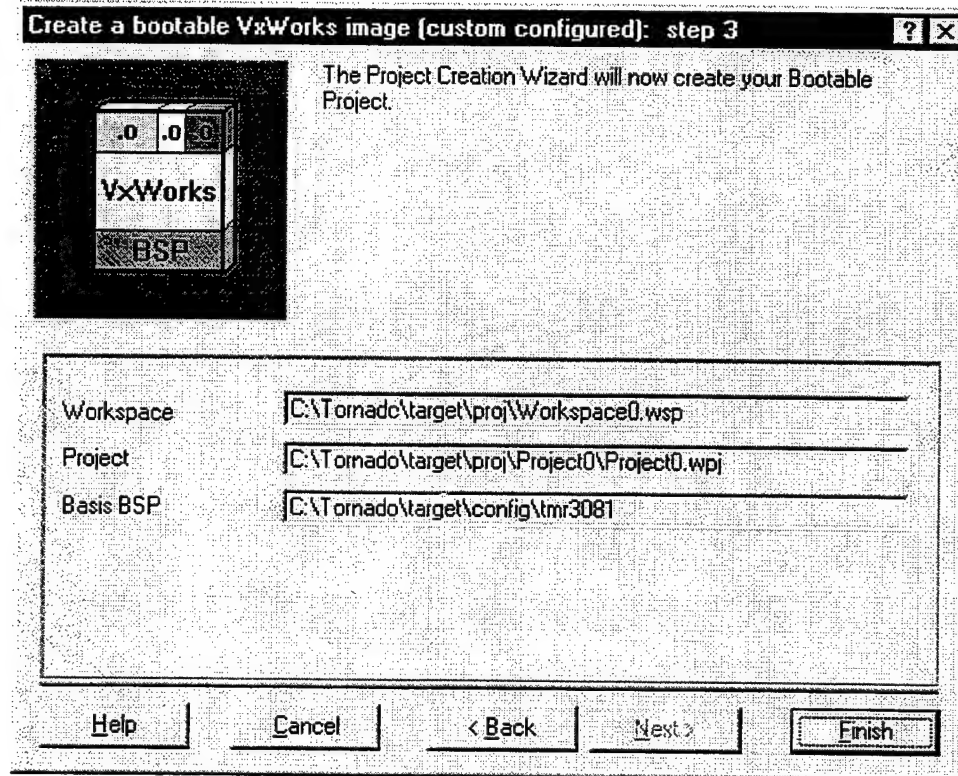
☒ Add to a New or Existing Workspace

Select a name for the Project, Location, Project description, and Workspace for the Project and hit the “Next” button. The rest of this appendix will assume the default names and locations were selected.

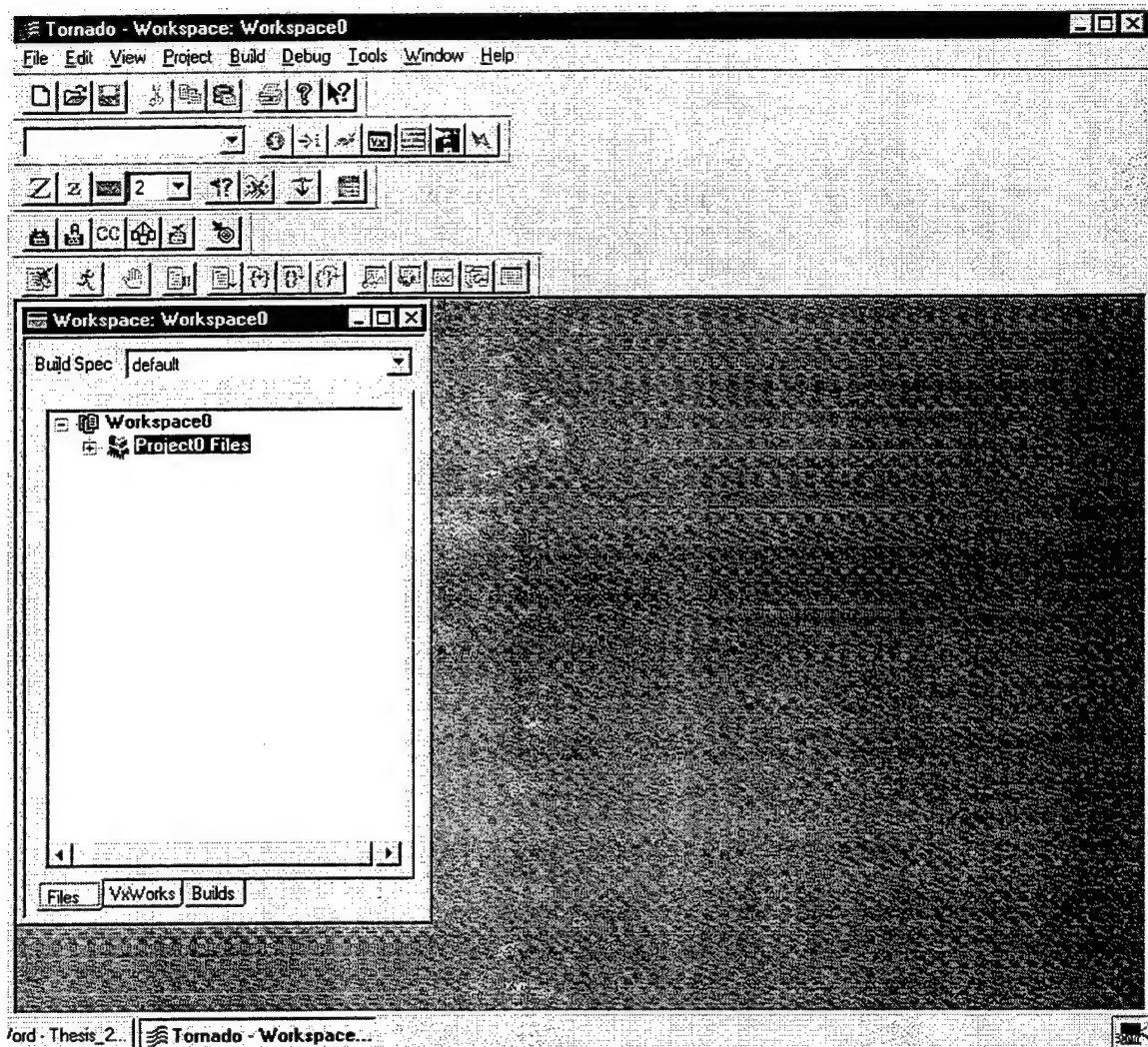
5. A window similar to the following window will appear. The “A BSP” radio button has been selected as well as the “tmr3081” directory. Make sure you make these changes and hit the “Next” button.



6. The following confirmation window will appear. Confirm your information and select the "Finish".



7. The New Project will be built and the following screen will appear:



You now have a project built using your BSP.

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF REFERENCES

Artesyn Technologies. Choosing an OS for Embedded Real-Time Application. 1998. Online. White Paper. Internet. 12 Dec 99. Available <http://www.artesyn.com/cp/html/choosingos.html>

Jun, Sun-Mi. Real Time Operating system in Embedded System and Case Study: pSOSystem Overview. 1998. Online. Internet. 12 Dec 99. Available [http://juno.cs.pusan.ac.kr/TechnicalReport/psos/rtos\\_psosystem.html](http://juno.cs.pusan.ac.kr/TechnicalReport/psos/rtos_psosystem.html)

Hawley, Greg. Selecting a Real-Time Operating System. 1999. Online. Embedded Systems Programming. Internet. 12 Dec 99. Available <http://www.embedded.com/1999/9903/9903sr.htm>

Hix, Deborah, and H. Rex Hartson. Developing User Interfaces: Ensuring Usability Through Product and Process. New York: Wiley, 1993.

Payne, John C. "Fault Tolerant Computing Testbed: A Tool For the Analysis of Hardware and Software Fault Handling Techniques." Naval Postgraduate School, 1998.

Schmorrow, Dylan D. "A Benchmark Usability Study of the Tactical Decision Making Under Stress Decision Support System". Naval Postgraduate School, 1998.

Sellers, J. J. Understanding Space, An Introduction To Astronautics. New York: McGraw-Hill, 1994.

Shneiderman, B. Designing the user interface: Strategies for effective human-computer interaction. Reading, MA: Addison-Wesley, 1997.

Stallings, William. Operating Systems: Internals and Design Principles. New Jersey: Prentice-Hall, 1998.

Storey, Neil. Safety-Critical Computer Systems. Reading, MA: Addison-Wesley, 1996.

Wind River Systems, Inc. Tornado™ User's Guide (Windows Version). 1<sup>st</sup> ed. 1999.

---. Tornado™ BSP Developer's Kit for VxWorks® User's Guide, Tornado 2.0. 1<sup>st</sup> ed. 1999.



THIS PAGE INTENTIONALLY LEFT BLANK

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..... 2  
 8725 John J. Kingman Road, Suite 0944  
 Ft. Belvoir, VA 22060-6218
  
2. Dudley Knox Library..... 2  
 Naval Postgraduate School  
 411 Dyer Road  
 Monterey, CA 93943-5101
  
3. Director, Training and Education ..... 1  
 MCCDC, Code C46  
 1019 Elliot Road  
 Quantico, VA 22134-5027
  
4. Director, Marine Corps Research Center ..... 2  
 MCCDC, Code C40RC  
 2040 Broadway Street  
 Quantico, VA 22134-5107
  
5. Marine Corps Representative ..... 1  
 Naval Postgraduate School  
 Code 037, Bldg. 330, Ingersoll Hall, Room 116  
 555 Dyer Road  
 Monterey, CA 93943
  
6. Marine Corps Tactical Systems Support Activity ..... 1  
 Technical Advisory Branch  
 Attn: Librarian  
 Box 555171  
 Camp Pendleton, CA 92055-5080
  
7. Chairman Code CS..... 1  
 Naval Postgraduate School  
 833 Dyer Road  
 Monterey, CA 93943

8. Professor Bret Michael ..... 1  
Naval Postgraduate School  
Code CS/Mj  
833 Dyer Road  
Monterey, CA 93943
9. Professor Alan Ross ..... 2  
Naval Postgraduate School  
Code SP/Ra  
833 Dyer Road  
Monterey, CA 93943
10. Professor Hersch Loomis ..... 1  
Naval Postgraduate School  
Code EC/Lm  
833 Dyer Road  
Monterey, CA 93943
11. LCDR Chris Eagle..... 1  
Naval Postgraduate School  
Code CS/Ce  
833 Dyer Road  
Monterey, CA 93943
12. LT Susan E. Groening ..... 3  
207 Remagen Road  
Seaside, CA 93955
13. Capt Kimberly D. Whitehouse, USMC..... 3  
18 Larkwood Court  
Stafford, VA 22554